IBM® Workplace Forms™ Server - API

Version 2.6

COM API User's Manual

S229-1528-00

Note
Before using this information and the product it supports, read the information in "Notices," on page 157.

First Edition (September 2006)

This edition applies to version 2.6 of IBM Workplace Forms Server - API (product number L-DSED-6JLR37) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Introduction

Welcome to the COM Edition of the user's manual for the IBM® Workplace Forms™ Server — API. The API extends the capabilities of Workplace Forms by enabling you to:

- Manipulate XFDL forms from new or existing applications.
- Create custom-built functions that may be integrated into XFDL forms.

This section discusses the organization and format of this manual. To learn more about the API, refer to "About the API" on page 3.

About This Manual

This manual has been organized as both an instruction manual and a quick reference. It describes the functions available in the API and provides examples of their use.

This manual contains the following major sections:

Section	Page
Introduction — introduces you to the features of the API.	"Introduction"
Overview of the Form Structure — explains how XFDL forms are stored in memory.	"Overview of the Form Structure" on page 5
Getting Started with the Form Library — provides a detailed tutorial demonstrating how to create a simple application that interacts with an XFDL form.	"Introduction to the Form Library" on page 13
Form Library Quick Reference — a reference to the functions contained in the Form Library. Each method description includes sample code.	"Form Library Quick Reference Guide" on page 23

Who Should Read This Manual

The API is designed to be easy to use for any moderately experienced programmer. However, the skill level required to develop particular functions may be quite high. This document is intended for developers who have a working knowledge of:

- COM compliant programming and syntax.
- Extensible Forms Description Language (XFDL) and syntax. Refer to the *Extensible Forms Description Language Specification* for more information.

Document Conventions

The following conventions appear throughout this manual:

- Sample code is presented in a monospaced font, and is indented to make the code stand out:

```
Sub SaveForm(Form, ThePath)
   ' Write the form to a file on disk
   Form.WriteForm ThePath, Nothing, 0
End Sub
```

- Text in bold italics represents information that you need to supply:

```
<label sid="firstName">
   <value>your first name here</value>
<label>
```

- The hash symbol (#) represents a number.
- Angle brackets enclose placeholders. For example, *<API Program Folder>* represents the actual folder in which you installed the API.
- Braces indicate optional items. The following example indicates that the item tag (including the period after it) is optional:

```
{itemtag.} option
```

- *"xx"* or *"xxx"* appears in place of the two or three digit version number of the API. In particular, these placeholders appear when referring to file names, folders, and directories that contain the API's version number.
- Brackets are used to indicate a sequence of choices, and the pipe symbol (|) is used to indicate "or". The following example indicates that you can use a number or a name:

```
(number|name)
```

About the API

The Workplace Forms Server — Application Programmer Interface (API) consists of a collection of programming tools to help you develop applications that can interact with XFDL forms. These tools are available for both C and Java programming environments. The API enables you to access and manipulate forms as structured data types.

The API is divided into two libraries: the Form Library and the Function Call Interface (FCI) Library. The Form Library allows you to create applications that:

- Read and write forms.
- Retrieve information from form elements.
- Add cells to certain form items.
- Insert information into form elements.

For more information about the Form Library refer to the "Form Library Quick Reference Guide" on page 23.

Where the API Fits in Your System

IBM provides a powerful suite of forms software for creating, using and transmitting forms over the Internet. The main components of this suite are:

Workplace Forms Viewer — Use the Viewer to view XFDL forms just as you would use a web browser to view HTML pages. You can also use the Viewer to fill out forms and submit them for review.

Workplace Forms Designer — The Designer provides an easy to use WYSIWYG design environment for creating XFDL forms. Use the Designer to create forms quickly and easily.

Workplace Forms API — The API is made up of a collection of Form functions. Use the Form Library to develop applications that manipulate XFDL forms.

Differences Between the C, Java, and COM Editions of the API

The various editions of the API differ in the following ways:

- The Java and COM editions offer an object-oriented interface.
- The COM edition does not support the FCI Library.
- The COM edition does not include the following Form Library functions:
  - GetInfoEx
  - GetAttributeList
- The COM edition includes the following Form library functions that the other editions do not:
  - GetType
  - GetIdentifier
  - ReadFormFromASPRequest
  - WriteFormToASPResponse

In all other respects, the different editions of the API provide the same functionality, and use the same memory model for forms.

The API Data Types

IFormNodeP Objects

The functions in the Form Library store forms in memory as a series of linked nodes. Each node, regardless of its level in the hierarchy, is represented by a IFormNodeP object.

The functions in the Form Library are responsible for creating and populating these nodes, and for freeing the memory they occupy.

About Memory Use

The Form methods are responsible for creating and populating these nodes. Furthermore, once you are done working with a form, you must use the **destroy** method on the root node of the form to remove it from memory.

About the API Constants

Several of the Form Library functions accept parameters whose values are constants (for example, the *referenceType* parameter of **DereferenceEx**). All constants have been defined in the IFS_COM_API type library, and are available by including that library in your source file, as shown:

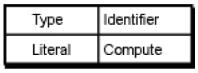
```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

Overview of the Form Structure

This section provides an overview of an XFDL form as it is represented in memory. Developers must understand the memory structure of a form to effectively develop applications using the API.

The Node Structure

When a form is loaded into memory, it is constructed as a series of linked nodes. Each node represents an element of the form, and together these nodes create a tree that describes the form. The following diagram illustrates the general composition of a single node.



Each node within the tree has the following properties:

- **Type** — For page and item nodes, this describes the type of node, such as *button*, *line*, *field*, and so on. Page nodes are always of type *page*.
- **Literal** — The literal value of the node (for example, a literal string). If the node has a formula, the result of the formula will be stored here.
- **Identifier** — The page tag, item tag, option name, or custom name assigned to the node.
- **Compute** — The compute assigned to the node (for example, ″field_1.value + field_2.value″). The result of the compute will be stored in the literal of the node.

Depending on the node type, some of these properties may be null.

The Node Hierarchy

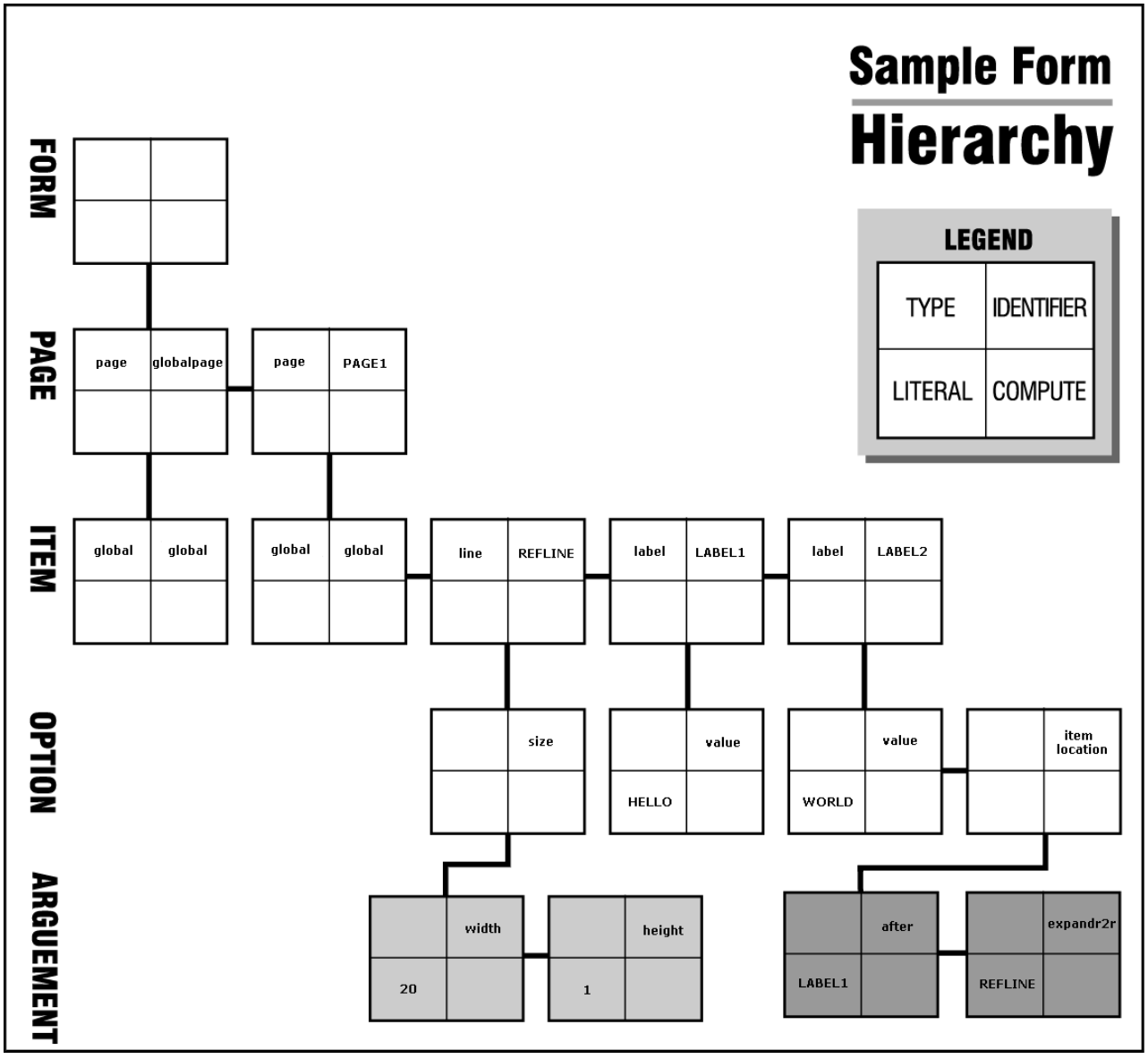
Every node is part of an overall hierarchy that describes the complete form. This hierarchy follows a standard tree structure, with the top of the tree being the top (or root) of the hierarchy.

The diagram on the following page illustrates the typical tree structure for a simple form.

The elements of the hierarchy, in descending order, are:

- **Form** — Each form has one form level node. This is the root node of the tree.
- **Page** — Each form contains pages, which are represented as children of the form node. Each form has at least two page nodes - one for the globalpage, which stores the global settings, and one for the first page of the form.
- **Item** — Each page contains items, which are represented as children of the page node. An item node is created for each item, including the global item which stores page settings.
- **Option** — Each item contains options, which are represented as children of the item node. An option node is created for each option.
- **Argument** — Options often contain further settings, or arguments, which are represented as children of the option node or as children of other argument

nodes. There may be more than one level of argument node created below an option node, depending on the option's settings. The easiest way to access a particular node in the hierarchy is to use a reference. References allow you to locate a specific node without first having to locate the parent of that node.



References

References allow you to identify a specific page, item, option, or argument by providing a "path" to that element. This means that you can access an element directly without having to locate any of its ancestors. The syntax of a reference follows this general pattern:

```
page.item.option[argument]
```

Each element of the reference is constructed as follows:

- **Page** and **Item** — Pages and items are identified by their scope identifiers (sid). For example, *Page1* or *Field1*.

- **Options** — Options are identified by their tag name. For example, *value* or *itemlocation*.
- **Arguments** — Arguments are identified by their tag name or a zero-based numeric index. Argument references are always enclosed in brackets. For example, *[1]* or *[message]*.

  Arguments can also have any depth. For example, you might have an argument that contains arguments. You can reference additional levels of depth by adding another bracketed reference. For example, to refer to the first argument in the first argument of the *printsettings* option, you could use either *[0][0]* or the tag names in brackets, such as *[pages][filter]*.

You can create references to any level of the node hierarchy. For example, the following table illustrates a number of references starting at different levels of the form:

Start At	Ref to Page	Ref to Item	Ref to Option	Ref to Argument
Page	Page1	Page1.Field1	Page1.Field1.format	Page1.Field1.format[message]
Item	—	Field1	Field1.format	Field1.format[message]
Option	—	—	format	format[message]
Argument	—	—	—	[message]

Dereferencing

When making a reference to an item node, there may be times when you do not know which node to reference because it depends on some action from the user of the form. Consider a situation in which a user selects a cell from a list. Because you don't know beforehand which cell the user will choose, it is not possible to explicitly reference the item node for the chosen cell. In such cases you would use *dereferencing* to retrieve the node indirectly.

Essentially, dereferencing allows you to make a dynamic reference that is evaluated at runtime. This is accomplished by placing the -> symbol to the right of the dynamic reference.

For example, consider a list item called *List1* that has three cells called *Cell1*, *Cell2*, *Cell3*. If you wanted to access the item node of the cell selected by the user, we would use the following reference string:

```
List1.value->
```

At runtime, the portion of the expression that is to the left of the dereference symbol is evaluated and replaced. If the user chose the second cell, *List1.value* would be evaluated and replaced with:

```
Cell2
```

As a result, the item node for *Cell2* would be returned.

In some cases, instead of accessing the item node of the chosen cell, you may want to access one of the cell's option nodes. Again, dereferencing is used. The reference string would be:

```
List1.value->value
```

As before, the above expression is evaluated at runtime. The expression to the left of the dereference symbol is evaluated and replaced, just as before. So if the second

cell was selected, *List1.value* would be evaluated as *Cell2*. This value is then concatenated with the expression to the right of the dereference symbol. This would produce:

```
Cell2.value
```

As a result, the option node for *Cell2.value* would be returned.

Note: Do not include any spaces before or after the dereference symbol (->).

Namespace in References

References that include options or arguments in any namespace other than XFDL normally require the inclusion of the namespace prefix in the reference. For example, if you were referencing ″myOption″ in the ″custom″ namespace, you would refer to that option as ″custom:myOption″ as shown:

```
page_1.myItem.custom:myOption
```

If you are referencing named arguments, you should also use the appropriate namespace. For example:

```
page_1.myItem.custom:myOption[custom:myArgument]
```

However, if you are referencing an argument by index number you do not need to worry about namespace. All arguments, regardless of namespace, are indexed in order. For example, if ″myOption″ contained two arguments, the first in the XFDL namespace and the second in the custom namespace, you would use the following reference for the second argument:

```
page_1.myItem.custom:myOption[1]
```

Note: Page and item references never require a namespace prefix because they are uniquely identified by their sid.

The null Namespace

In some cases, forms may have no default namespace or may have a default namespace that is explicitly set to an empty string. In these cases, you can use *null* as the prefix for the empty namespace. For example, the following field declares a default namespace that is empty:

```
<page sid="Page1">
   <field sid="myField" xmlns="">
      <value>Test Value</value>
   </field>
</page>
```

In this case, to reference the value of the field, you would use the null prefix as shown:

```
Page1.null:myField.null:value
```

Advanced Information about the Node Structure

When an XFDL form is stored in memory, it exists as a series of nodes that are linked in a tree structure. As described in “The Node Hierarchy” on page 5, the tree structure follows this hierarchy: form, page, item, option, and argument.

Within a single branch of the tree, all elements of the same level are treated as siblings, each of which has a common parent, and each of which may have its own children.

The following example illustrates the node structure of a simple form, and gives a top-down description of the node structure.

A Sample Hierarchy

The following XFDL code creates the node hierarchy shown in . The result is a simple form that contains three items (a line and two labels).

```
<?xml version = "1.0"?>
<XFDL xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
   xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0">
   <globalpage sid="global">
      <global sid="global"></global>
   </globalpage>
   <page sid = "PAGE1">
      <global sid="global"></global>

      <line sid = "REFLINE">
         <size>
            <width>20</width>
            <height>0</height>
         </size>
      </line>
      <label sid = "LABEL1">
         <value>Hello</value>
      </label>
      <label sid = "LABEL2">
         <value>World</value>
         <itemlocation>
            <after>LABEL1</ae>
            <expandr2r>REFLINE</expandr2r>
         </itemlocation>
      </label>
   </page>
</XFDL>
```

The Sample Tree Structure

Each tree begins with the form, or root, node. This node contains no information - it simply represents the starting point of the tree structure.

Below the form node are the page nodes. In the previous example, there are two page nodes: ″global″ and ″PAGE1″. The ″global″ page node stores any global settings that apply to the form while ″PAGE1″ stores the contents of the first form page. Any additional pages would also be stored as children of the form node.

Below each page node are the item nodes. As illustrated in the previous example, the first item node for any page is always the ″global″ item. The ″global″ item stores any page settings that are applied to the items in that page. Each additional item in the page is stored as a sibling of the global item.

Note: The ″global″ page node will always have one child: the global item. This global item will always store the XFDL version number used to create the form, and is also used to store any global settings that are applied to the form.

Below each item node are the option nodes. Each option node represents an option setting for that item, such as a background color or font setting.

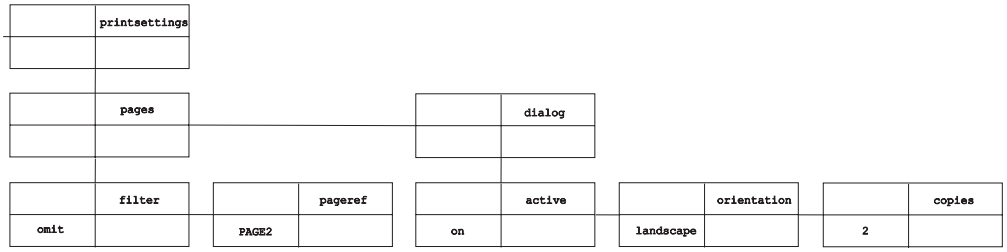
Below each option node are the argument nodes. These nodes contain the settings for the parent option. For example, the background color might be set to ″blue″.

There can be an infinite number and depth of these nodes, depending upon the number and depth of the settings for that option.

For instance, in the sample form, the *size* node for "REFLINE" has two argument nodes: one for the width and one for the height. In contrast, the *printsettings* option can have multiple argument nodes which themselves have argument nodes as children. The following is an example of the node structure of the *printsettings* option:

```
<printsettings>
   <pages>                                  argument node level I
      <filter>omit</filter>                 argument node level II
      <pageref>page2</pageref>              argument node level II
   </pages>
   <dialog>                                 argument node level I
      <active>on</active>                   argument node level II
      <orientation>landscape<orientation>   argument node level II
      <copies>2</copies>                    argument node level II
   </dialog>
</printsettings>
```

printsettings Node Structure



Thus, in storing the *printsettings* option, two levels of argument nodes are created. The first level describes the number of array elements in the option (two). The second level gives the arguments for each element.

Due to their potential complexity, pay careful attention to the mapping of argument nodes.

Note: In cases where an option has multiple elements in an array (for example, *printsettings*), there will be a single option node, but a separate argument node for each element in the array.

Node Properties

There are several levels of nodes in an XFDL form: form (or root), page, item, option, and argument (which can have an infinite number of levels). Each node has four properties: literal, type, identifier, and compute. A node does not necessarily contain information for every property.

For example, a page node can never have values for the compute or literal properties. And while a value for the user data property is optional, a page node must always have values for the type and identifier properties.

The following table illustrates what properties may be in use for each node level.

	Node Property			
Level	**Literal**	**Type**	**Identifier**	**Compute**
Form	no	no	no	no
Page	no	always	always	no
Item	no	always	always	no
Option	yes	no	always	yes
Argument (at any level)	yes	no	yes	yes

yes — node can have that property
always — node always has that property
no — node cannot have that property

Introduction to the Form Library

The Form Library is a collection of functions for developing applications that manipulate XFDL forms. Using the functions in the Form Library, your applications can:

- Read and write forms.
- Retrieve information contained in a form's elements.
- Assign information to the elements of a form.
- Create new elements within a form.
- Remove elements from a form.
- Extract images or enclosures from a form.
- Verify digital signatures.

Essentially, an XFDL form may be thought of as a structured data type, with the API as the means for accessing this data structure.

Getting Started with the Form Library

This section provides a detailed tutorial to help you understand how to use the Form Library. By working through the tutorial, you will perform all of the steps involved in creating a simple application that uses the API functions, including:

- Initializing the Form Library.
- Reading a form into your application.
- Setting and retrieving form data.
- Removing a form from memory.

The sample application in this tutorial reads an input form called CalculateAge.xfd into memory. It retrieves the user's birth day, month, and year as well as the current date from the form. It then places these values into hidden fields in the form. This triggers the form to compute the user's age and display the result. When complete, the application saves the changes made to calculateAge.xfd as a new form called Output.xfd.

Note: The sample application described in this tutorial is included with the API and can be found in the folder: <*API Program Folder*>\Samples\COM\Form\Demo\Calculate_Age\

The tutorial describes the following tasks:

Note: Before you can build applications using the Form Library, you must install the API and set up your development environment. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information.

Setting Up Your Application

This example assumes that you are developing a program in Microsoft Visual Basic using Microsoft Studio. To begin, you must first set up your development environment to use the API type library.

1. Create a new Visual Basic project.
2. Any project using the API must include the following type library:
   - IFS_COM_API.tlb

To add this to your Visual Basic Project, open the **References** dialog from the **Project** menu and select "InternetForms API".

3. Set up the rest of your application. This generally includes setting up your main algorithm and declaring any variables you will need. The following code sets up the Calculate Age application:

```
' Create the Main method for the program.

Sub Main()

   ' Declare a number of variables. TheForm represents the form, while the
   ' other variables are values we will read from the form.

   Dim TheForm As IFormNodeP
   Dim BirthYear As Integer
   Dim BirthMonth As Integer
   Dim BirthDay As Integer

   ' The program's Main consists of a number of calls to other functions.

   Initialize

   Set TheForm = LoadForm

   BirthYear = GetBirthYear(TheForm)
   BirthMonth = GetBirthMonth(TheForm)
   BirthDay = GetBirthDay(TheForm)

   SetBirthYear BirthYear, TheForm
   SetBirthMonth BirthMonth, TheForm
   SetBirthDay BirthDay, TheForm

   SaveForm TheForm

   ' Free the memory in which the form was stored.

   TheForm.Destroy

End Sub
```

Initializing the Form Library

All applications that use the API functions must initialize the Form Library to ensure correct error and memory handling behavior. The sample application does this in a separate method called **Initialize**. In turn, **Initialize** calls the Form Library function **IFSInitialize** and passes it the name of the current program.

Define the **Initialize** function to call the function **IFSInitialize**. **IFSInitialize** initializes the API environment.

```
Sub Initialize()

   Dim DTK As DTK

   ' Get the DTK object. You need this call the IFSInitialize function.

   Set DTK = CreateObject("PureEdge.DTK")

   ' Call DTK.IFSInitialize.  DTK is a static class representing the Java

   ' API development toolkit.  The parameters are:
   '   1. CalculateAge: the name of the application being run.
   '   2. 1.0.0 : the version of the application being run.
   '   3. 2.6.0 : the version of the API being run.
```

```
    ' An exception will be thrown if there is a problem.

    DTK.IFSInitialize "CalculateAge", "1.0.0", "2.6.0"

End Sub
```

Note: For detailed information about the **IFSInitialize** function, including a description of its parameters, refer to "IFSInitialize" on page 117.

Loading a Form

Before your program can begin working with a form, you must load it into memory. CalculateAge does this by defining a **LoadForm** function to handle these tasks.

Call **ReadForm** within the implementation of your **loadForm** function to read in the form, calculateAge.xfd.

```
Function LoadForm() As IFormNodeP

    Dim XFDL As XFDL

    ' Get the XFDL object. You need this to call the ReadForm function.
    Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
    ' Call XFDL.ReadForm.  The parameters are:
    '   1. calculateAge.xfd : indicates the file on the local drive to read
    '       the form from.
    '   2. 0 : no special behavior.
    '   readForm will return a reference to the root node of the
    '   new form structure once it is loaded into memory.  An exception
    '   will be thrown if there is a problem.

    Set LoadForm = XFDL.ReadForm("c:\calculateage.xfd", 0)

End Function
```

Note: For more information about the **ReadForm** function, refer to "ReadForm" on page 152.

Retrieving A Value from a Form

Once you have set up and initialized your application with the API and loaded a form into memory, your application is ready to start working with the form. The following code uses **GetLiteralByRefEx** to get a specific value from the form:

Retrieve the birth date from the form as three separate values: the birth day, birth month, and birth year. To do this, we will use three separate functions. Each function uses **GetLiteralByRefEx** to retrieve one of the values.

```
Function GetBirthDay(TheForm) As Integer

    Dim BDay As String

    ' Call IFormNodeP.GetLiteralByRefEx to get the literal information for
    ' the PAGE1.BIRTHDAY.value node.  An exception will be thrown if
    ' there is a problem.

    BDay = TheForm.GetLiteralByRefEx(vbNullString, _
       "PAGE1.BIRTHDAY.value",  0, vbNullString, Nothing)

    ' If a literal value was returned, convert it into an integer value;
    ' otherwise, indicate that no value was entered into the field
```

```
      ' and throw an exception.

      If (Len(BDay) > 0) Then
         GetBirthDay = CInt(BDay)
      Else
         MsgBox "The birth day was not entered.", vbCritical
         Stop
      End If

   End Function

' Similar to GetBirthDay

   Function GetBirthMonth(TheForm) As Integer

      Dim BMonth As String

      BMonth = TheForm.GetLiteralByRefEx(vbNullString, _
         "PAGE1.BIRTHMONTH.value", 0, vbNullString, Nothing)

      If (Len(BMonth) > 0) Then
         GetBirthMonth = CInt(BMonth)
      Else
         MsgBox "The birth month was not entered.", vbCritical
         Stop
      End If

   End Function

' Similar to getBirthDay()

   Function GetBirthYear(TheForm) As Integer

      Dim BYear As String

      BYear = TheForm.GetLiteralByRefEx(vbNullString, _
         "PAGE1.BIRTHYEAR.value", 0, vbNullString, Nothing)

      If (Len(BYear) > 0) Then
         GetBirthYear = CInt(BYear)
      Else
         MsgBox "The birth year was not entered.", vbCritical
         Stop
      End If

   End Function
```

Note: For detailed information about the **GetLiteralByRefEx** method, including a description of its parameters, refer to "GetLiteralByRefEx" on page 58.

Setting a Value in a Form

Once a form is loaded into memory, a developer can set the values associated with any of the item or option nodes located in the form by calling **SetLiteralByRefEx**.

Change the values of hidden fields in the original form. These hidden fields are arguments for a compute in the SHOWAGE label that calculates the user's age. To do this, we will use three separate functions. Each function uses **SetLiteralByRefEx** to set one of the values.

```
   Sub SetBirthDay(BDay As Integer, TheForm As IFormNodeP)

      Dim Day As String
```

```
    ' Convert the birthday to a String

    Day = CStr(BDay)

    ' Call TheForm.SetLiteralByRefEx. The parameters are:
    '   1. vbNullString : reserved. Must be vbNullString.
    '   2. PAGE1.HIDDENDAY.value : the reference to the node.
    '   3. 0 : must be zero.
    '   4. vbNullString : sets the character set to ANSI.
    '   5. Nothing : no namespace node required.
    '   6. Day : the value to assign to the literal.
    ' An Exception will be thrown if there is a problem.

    TheForm.SetLiteralByRefEx vbNullString, "PAGE1.HIDDENDAY.value", _
       0, vbNullString, Nothing, Day

End Sub

' Similar to SetBirthDay()

Sub SetBirthMonth(BMonth As Integer, TheForm As IFormNodeP)

    Dim Month As String

    ' Convert the birth month to a String

    Month = CStr(BMonth)

    ' Set the birth month as a value in the form.
    TheForm.SetLiteralByRefEx vbNullString, "PAGE1.HIDDENMONTH.value", _
       0, vbNullString, Nothing, Month

End Sub

' Similar to setBirthDay()

Sub SetBirthYear(BYear As Integer, TheForm As IFormNodeP)

    Dim Year As String

    ' Convert the birth year to a String

    Year = CStr(BYear)

    ' Set the birth year as a value in the form.

    TheForm.SetLiteralByRefEx vbNullString, "PAGE1.HIDDENYEAR.value", _
       0, vbNullString, Nothing, Year

End Sub
```

Note: For detailed information about **SetLiteralByRefEx**, including a description of its parameters, refer to "SetLiteralByRefEx" on page 92.

Writing a Form to Disk

Once you have finished making the desired changes to the form, you should save it to disk. If you want to retain the original form (calculateAge.xfd), you should save the modified form under a new name. This program saves the modified form as Output.xfd.

The following example implements the function **SaveForm**. This function calls the API function **WriteForm** that writes a form to disk.

```
Sub SaveForm(TheForm)

   ' Call theForm.writeForm.  theForm is the root node of the form to
   ' be written.  The parameters are:
   '   1. output.xfd : the filename you want to use (you could also use
   '        a path here).
   '   2. Nothing : since we do not want to set a triggeritem.
   '   3. 0 : since we do not want to allow the transmit options to work.
   ' An exception is thrown if there is a problem.

   TheForm.WriteForm "output.xfd", Nothing, 0

End Sub
```

Note: For detailed information about **WriteForm**, including a description of its parameters, refer to "WriteForm" on page 109.

Closing a Form

Next, you must free the memory used by the form itself. This is the last operation in the main function of the program.

The program's main method calls the API's **Destroy** method to delete TheForm object.

```
   ' Now that we are done with the form, we can free the memory by
   ' calling Destroy.  The object, 'TheForm', is a reference to
   ' the root node of the form.  This causes the root node and all
   ' of its children (the complete form) to be deleted from memory.

   TheForm.Destroy

End Sub
```

Note: For detailed information about **Destroy**, including a description of its parameters, refer to "Destroy" on page 44.

Compiling Your Application

Once you have generated the source files for your application, you must compile the source code.

- Use an appropriate compiler that is supported by this API to compile your files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for more information about compatible development environments.
- Before building your application you should have a source file that represents your application. After compiling the file you will have a file an executable (or class file) with the same name.
- The details of compiling your source code are not included in this manual. Consult your development environment's documentation for specific information on how to use your compiler.
- Make sure that the compiler uses the -I option when searching the directory containing the API include files.

Testing your Application

Use the sample form that accompanies the API to test the Calculate Age application.

1. Copy the file calculateAge.xfd to the folder containg your application. The file is located in the following folder:

   `<API Program folder>\Samples\COM\Form\Demo\Calculate_Age\`
2. Open the form in the Viewer to see the original settings.
3. Run the application that you have just created.
4. A new file will be created called Output.xfd.

Note: To view the forms provided with this API, you must have a licensed or evaluation copy of the IBM Workplace Forms Viewer installed.

Distributing Applications That Use the Form Library

If you distribute applications that use the Form Library, you will also need to distribute a number of API files. Refer to the *IBM Workplace Forms Server - API Installation and Setup Guide* for information about distributing applications that use the Form Library.

Summary

By working through this section you have successfully built the Calculate Age application. In the process, you have learned how to initialize, compile, and test form applications using the following methods from the Form Library:

- IFSInitialize
- ReadForm
- GetLiteralByRefEx
- SetLiteralByRefEx
- WriteForm
- Destroy

The source code for the Calculate Age application is included with this API and can be found in the following folder:

`<API Program folder>\Samples\COM\Form\Demo\Calculate_Age\`

To view the forms provided with the sample application, you must have a copy of the Viewer installed.

Note: The sample files provided are compatible with the Microsoft Visual Basic 6.0 development environment. For more information about compatible development environments for the API refer to the *IBM Workplace Forms Server - API Installation and Setup Guide.*

Form Library Quick Reference Guide

This section provides detailed information about the Form Library. The available functions are divided into the following categories:

Within each section, the functions are presented alphabetically.

Form Library Functions

The Form Library includes the following functions:

Function Type	Description	Functions
Certificate	The **Certificate** function gets information from digital certificates.	GetBlob GetDataByPath GetIssuer
FormNodeP	The **FormNodeP** functions create and populate nodes and free memory.	AddNamespace CreateCell DeleteSignature DereferenceEx Destroy Duplicate EncloseFile EncloseInstance

Function Type	Description	Functions
FormNodeP (continued)	The **FormNodeP** functions create and populate nodes and free memory.	ExtractFile ExtractInstance ExtractXFormsInstance GetAttribute GetAttributeList GetCertificateList GetChildren GetFormVersion GetIdentifier GetLiteralEx GetLiteralByRefEx GetLocalName GetNamespaceURI GetNamespaceURIFromPrefix GetNext GetNodeType GetParent GetPrefix GetPrefixFromNamespaceURI GetPrevious GetReferenceEx GetSecurityEngineName GetSigLockCount GetSignature GetSignatureVerificationStatus GetType IsSigned IsXFDL

Function Type	Description	Functions
FormNodeP (continued)	The **FormNodeP** functions create and populate nodes and free memory.	RemoveEnclosure ReplaceXFormsInstance SetActiveForComputationalSystem SetAttribute SetFormula SetLiteralEx SetLiteralByRefEx SignForm ValidateHMACWithSecret ValidateHMACWithHashedSecret VerifyAllSignatures VerifySignature WriteForm WriteFormToASPResponse XMLModelUpdate
Hash	The **Hash** functions hash data.	Hash
Initialization	The **Initialization** functions initialize the API.	IFSInitialize
LocalizationManager	The **LocalizationManager** functions control which locale (language) the API uses.	GetCurrentThreadLocale GetDefaultThreadLocale SetCurrentThreadLocale SetDefaultLocale
SecurityManager	The **SecurityManager** functions provide hashing algorithms.	LookupHashAlgorithm
Signature	The **Signature** functions get information from signatures.	GetDataByPath GetSigningCert
XFDL	The **XFDL** functions create the root nodes of XFDL forms and handle administrative tasks related to the API.	Create GetEngineCertificateList IsDigitalSignaturesAvailable ReadForm ReadFormFromASPRequest

About the Function Descriptions

The functions in this reference guide are listed according to the functionality they provide and are described using the following format:

- **Description:** Provides a general description of what the function does.
- **Function:** Lists the function's signature and type of value returned (if any).
- **Parameters:** Lists and describes each parameter in detail.
- **Returns:** Indicates what value is returned by the function.
- **Notes:** Provides additional information to help you use the function.
- **Example:** Provides sample code that uses the function in question.

Note: All sample code for the COM API is written in VBScript, and assumes it is running as part of an ASP page. However, you can use any COM compliant language when writing your own programs.

Using Signatures with the Form Library

Computed options often contain their current computed value. If this value is signed, it will not change, even if something in the form changes that would normally trigger the compute.

The literal value is stored as simple character data in the computed option, as shown below:

```
<field sid="FIELD1">
   <value compute="page1.nameField.value">Jane E. Smith</value>
</field>
```

The node structure for this *value* option is:

field	FIELD1
	value
Jane E. Smith	Page1.nameField.value

The Viewer sets this literal value when a form is signed, submitted, or saved (and discards any old value if necessary). When **ReadForm** is invoked, the current value (cval) is set and cannot be changed. Because a digitally signed formula never fires after being signed, the current value for the option is always the same - and therefore it is possible to reference the option and get the signed literal value.

About Errors

The COM API reports errors in one of two ways, depending on the programming language you are using. If your programming language supports structured errors, the COM API will throw exceptions. Otherwise, the COM API will report errors in the HResult, and you will have to handle those errors manually.

About Output Parameters

When using VBScript, the COM interface does not properly match output parameters with variant data types. This means that any function requiring an output parameter will not function properly. If you are programming in VBScript, set all output parameters to **null** rather than using a variable.

The nulls required under VBScript are:

- Nothing — Used for all objects.
- vbNullString — Used for all strings.
- vbNull — Used for other data types. vbNull is equivalent to the constant ″1″, and is not declared by default in VBScript. Rather than declaring a constant variable, you should use ″1″ in all cases.

The Certificate Functions

The **Certificate** functions allow you to work with *Certificate* objects.

- To use the Certificate functions you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

GetBlob

Description

This function extracts a binary long object (Blob). This Blob is a DER-encoded certificate.

Function

```
Function GetBlob(
   theStatus As Long
   ) As Variant
```

Parameters

Expression	Type	Description
theStatus	**Long**	A long that is set with the status of the operation. This will be one of the following: **SUSTATUS_OK** — The operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

The Blob as a variant containing an array of bytes.

Example

The following function extracts the Blob from a certificate, checks the status to make sure the operation was successful, then returns the Blob.

```
Sub extractBlob(TheCert)

   Dim TheBlob  ' Variant

   ' Get the Blob from the certificate

   TheBlob = TheCert.GetBlob(1) ' vbNull

   ' Call the ProcessBlob function to process the certificate Blob. Note
   ' that this is not an API function, but rather a function you would
```

```
    ' write to do some processing, such as writing the Blob to disk.

    ProcessBlob(TheBlob);

End Sub
```

GetDataByPath

Description

This function retrieves a piece of data from a certificate object.

Function

```
Function GetDataByPath(
   dataPath As String,
   tagData As Boolean,
   encoded As Boolean
   ) As String
```

Parameters

Expression	Type	Description
thePath	**String**	The path to the data you want to retrieve. See the Notes section below for more information on data paths.
tagData	**Boolean**	True if the path should be prepended to the data, or False if not. If the path is prepended, an equals sign (=) is used as a separator. For example, suppose the path is ″Signing Cert: Issuer: CN″ and the data is ″IBM″. If True, the path will be prepended, producing ″CN=IBM″. If False, the path will not be prepended, and the result will be ″IBM″.
encoded	**Boolean**	True if the return data is base 64 encoded, or False if not. The function returns binary data in base 64 encoding.

Notes

About Data Paths

Data paths describe the location of information within a certificate, just like file paths describe the location of files on a disk. You describe the path with a series of colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

For example, to retrieve the version of a certificate, you would use the following data path:

```
version
```

However, to retrieve the subject's common name, you first need to locate thesigning certificate, then the subject, then the common name within the subject, as follows:

```
SigningCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-based element number.

For example, the following path would retrieve a comma separated list:

```
Issuer: OU
```

While adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
Issuer: OU: 0
```

Certificate Tags

The following table lists the tags available in a certificate object:

Tag	Description
Subject	The subjects distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.
Issuer	The issuer's distinguished name. This is an object that contains further information, as detailed in *Distinguished Name Tags*.
IssuerCert	The issuer's certificate. This is an object that contains the complete list of certificate tags.
Engine	The security engine that generated the certificate. This is an object that contains further information, as detailed in *Security Engine Tags*.
Version	The certificate version.
BeginDate	The date on which the certificate became valid.
EndDate	The date on which the certificate expires.
Serial	The certificates serial number.
SignatureAlg	The signature algorithm used to sign the certificate.
PublicKey	The certificates public key.
FriendlyName	The certificates friendly name.

Distinguished Name Tags

The following table lists the tags available in a distinguished name object:

Tag	Description
CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

Security Engine Tags

The following table lists the tags available in the security engine object:

Tag	Description
Name	The name of the security engine used by the server.
Help	The help text for the security engine.
HashAlg	A hash algorithm supported by the security engine.

Returns

A string containing the certificate data (null if no data is found), or throws an exception if an error occurs.

Example

The following function uses **DereferenceEx** to locate a signature button in the form. It then calls **GetCertificateList** to get a list of valid certificates for that button. The function then loops through the available certificates, using **GetDataByPath** to check the common name of each certicate. When it finds the certificate with the common name of ″TJones″, it calls **SignForm** and uses that certificate to sign the form.

```
Sub ApplySignature(Form)

   Dim SigNode, SigObject  ' objects
   Dim TheCerts  ' CertificateList
   Dim CommonName  ' String
   Dim Cert  ' ICertificate

   ' Get the SignatureButton node

   Set SigNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.SignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get available certificates for that button

   Set TheCerts = SigNode.GetCertificateList(vbNullString, 1) 'vbNull

   ' Test each of the available certificates to see if it has a common
   ' name of "TJones".  If it does, use that certificate to sign
   ' the form.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      Response.Write CommonName & vbCrLf
      If CommonName = "TJones" Then
         Set SigObject = SigNode.SignForm(TheCerts(1), Nothing, 1)
            ' vbNull
      End If
   Next

End Sub
```

GetIssuer

Description

This function extracts the issuer certificate from the certificate provided.

Function

```
Function GetIssuer(
   theStatus As Long
   ) As Certificate
```

Parameters

Expression	Type	Description
theStatus	**Long**	A long that is set with the status of the operation. This will be one of the following: **SUSTATUS_OK** — The operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

The issuer certificate.

Example

The following example gets the signing certificate from a signature object, then iterates through the certificate issuers until it reaches the end of the chain. During the iteration, each certificate is passed to a function that processes them.

```
Sub processCertChain(TheSig)

   Dim TheCert, IssuerCert  ' ICertificate

   ' Get the signing certificate from the signature
   Set TheCert = TheSig.GetSigningCert()
   ' Loop through the certificate chain, passing each certificate to the
   ' ProcessCert function. The loop ends when the issuer certificate is
   ' Nothing.

   Do While (Not(TheCert Is Nothing))

      ' Pass the certificate to the ProcessCert function.  Note that
      ' this is not an API function, but rather a function you would
      ' write to process the certificate in some way.

      ProcessCert(TheCert)

      ' Get the issuer certificate from the TheCert

      Set IssuerCert = TheCert.GetIssuer(1) ' vbNull

      ' Assign theCert to equal the issuerCert for next iteration of the
      ' loop.
```

```
        Set TheCert = IssuerCert
    Loop
End Sub
```

The FormNodeP Functions

The **FormNodeP** functions apply to particular instances of a form and the items in that form.

- Each node in a form, regardless of its level in the node hierarchy, is represented by a **IFormNodeP** object. For more information about the node structure of XFDL forms refer to "Overview of the Form Structure" on page 5.
- To use the **FormNodeP** functions you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

FormNodeP Constants

The following table lists the constants that are used by the IFormNodeP functions along with a short description of each constant:

Named Constants	Description
UFL_DS_CERTEXPIRED	The certificate has expired.
UFL_DS_CERTNOTFOUND	The certificate was not found.
UFL_DS_CERTNOTTRUSTED	The certificate is no longer trusted.
UFL_DS_CERTREVOKED	The certificate has been revoked.
UFL_DS_F2MATCHSIGNER	The name in the form did not match the name in the signature.
UFL_DS_HASHCOMPFAILED	The hash of the document did not match the hash in the signature.
UFL_DS_ISSUERNOTFOUND	The issuer could not be found.
UFL_DS_ISSUERSIGFAILED	The verification of the issuer's certificate failed.
UFL_DS_SIGNATUREALTERED	The signature has been altered.
UFL_DS_UNEXPECTED	Generic error.

addNamespace

Description

This function adds a namespace declaration to the node it is called on. Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
Sub AddNamespace(
   theURI As String,
   thePrefix As String)
```

Parameters

Expression	Type	Description
theURI	**String**	The namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
thePrefix	**String**	The prefix for the namespace. For example, *xfdl*.

Returns

Nothing or throws an exception if an error occurs.

Example

The following function adds a *custom* namespace to a form and then adds a *custom* option to the global item. First, the function uses **AddNamespace** to add the *custom* namespace to the form. It then uses **DereferenceEx** to locate the global item on the form's global page. Finally, it uses **Create** to add a *custom* option to the global item.

```
Sub AddCustomNamespace(Node)

   Dim TempNode, XFDL  ' objects

   Set TempNode = Node

   ' Add the Custom namespace to the form

   TempNode.AddNamespace"http://www.ibm.com/xmlns/prod/XFDL/Custom", "custom"

   ' Locate the global item in the global page
   Set TempNode = TempNode.DereferenceEx(vbNullString, "global.global", _
      0, UFL_ITEM_REFERENCE, Nothing)

   ' Create an XFDL object so that we can use the create function

   Set XFDL = CreateObject("Workplace_Forms.xfdl_XFDL")

   ' Create a "status" option in the custom namespace as a child of the
   ' global item.

   Set TempNode = XFDL.Create(TempNode, UFL_APPEND_CHILD, vbNullString, _
      "Processed" , vbNullString, "custom:status")

End Sub
```

CheckValidFormats

Description

This function checks the format of all items in the form and returns the number of items whose format is invalid. You can also set the function to create a list of the invalid items.

This function does not support XForms nodes.

Function

```
Function CheckValidFormats() As Boolean
```

Parameters

There are no parameters for this function.

Returns

An array of nodes that represent items with invalid formats.

Example

The following function walks through the form and uses **CheckValidFormats** to determine which nodes have the correct format. This function assumes that you are passing in the root node of the form.

```
Sub CheckFormats

   Dim theForm ' object
   Dim invalidItems ' IFormNodePList
   Dim theItem ' IFormNodeP
Dim theReference ' String

   invalidItems = theForm.checkValidFormats();
   If invalidItems == Nothing Then
      Response.Write("All the items have valid formats."
   Else
      For Each theItem in invalidItems
         theReference = theItem.GetReferenceEx(vbNullString, Nothing, _
            Nothing, False)
      Response.Write("The item "theReference" has an invalid format".)

   End If

End Sub
```

createCell

Description

Use this function to create a new cell item for a *combobox*, *list*, or *popup*. **CreateCell** adds one new cell to a specific *group* on a specific page in the form. Note that this function can only assign a name to the new cell; it cannot set the cell's *value*. To set the value of a cell, you must use the **SetLiteralByRefEx** function.

This function is called from a page level node, and creates the new cell in that page. Note that you cannot call this function from the global page node.

Function

```
Function CreateCell(
   theCellName As String,
   theGroupName As String
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
theCellName	**String**	The name of the new cell being created.
theGroupName	**String**	The name of the *group* option to which the new cell will be added.

Returns

An **IFormNodeP** containing the new cell or throws an exception if an error occurs.

Example

The following function adds three cells to a form. First, the function uses **DereferenceEx** to locate the PAGE1 node. Next, the function calls **CreateCell** to create a new cell on that page and **SetLiteralByrefEx** to set the value of the new cell. The function then repeats these steps to create two more cells on that page.

```
Sub AddCells(Form)

   Dim PageNode, CellNode  ' object

   Set PageNode = Form

   Set PageNode = PageNode.DereferenceEx(vbNullString, "PAGE1", 0, _
      UFL_PAGE_REFERENCE, Nothing)

   Set CellNode = PageNode.CreateCell("RedCell", "AvailableColors")
   CellNode.SetLiteralByRefEx vbNullString, "PAGE1.RedCell.value", 0, _
      vbNullString, Nothing, "Red"

   Set CellNode = PageNode.CreateCell("BlueCell", "AvailableColors")
   CellNode.SetLiteralByRefEx vbNullString, "PAGE1.BlueCell.value", 0, _
      vbNullString, Nothing, "Blue"

   Set CellNode = PageNode.CreateCell("GreenCell", "AvailableColors")

   CellNode.SetLiteralByRefEx vbNullString, "PAGE1.GreenCell.value", 0, _
      vbNullString, Nothing, "Green   "

End Sub
```

DeleteSignature

Description

This function deletes the specified digital signature in the form. For security reasons, the form must meet certain criteria before this is allowed. None of the following should be locked by another signature: the signature, its descendants, the associated signature button, and its signer option. If these criteria are met, then the signature's locks are removed, and the signature item is deleted. Then, and the signer of the associated signature button is set to empty ("").

Function

```
Sub DeleteSignature(
   signatureItem As IFormNodeP)
```

Parameters

Expression	Type	Description
signatureItem	**IFormNodeP**	The signature node to delete.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Notes

If the *signature* item contains a *layoutinfo* option, **DeleteSignature** will not remove the entire signature from the form. Instead, the *signature* item and the *layoutinfo* option will remain. To completely delete the signature item, you must delete the remaining nodes manually by using **Destroy** to delete the signature item.

Example

The following function checks to see the signature in the form is valid. First, the function uses **DereferenceEx** to locate the signature button. It then uses **GetLiteralByRefEx** to get the name of the signature item, and uses another **DereferenceEx** to locate that item. Next, it uses **VerifySignature** to determine whether the signature is valid. If so, it return the string ″Valid″. If not, it uses **DeleteSignature** to delete the signature and returns the string ″Invalid″.

```
Function CheckSignature(Form)

   Dim TempNode, SigNode  ' objects
   Dim SigStatus  ' Integer
   Dim SigItemRef  ' Strings

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
      "PAGE1.SignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get a reference to the signature item from the signature option

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Verify the signature

   SigStatus = Form.VerifySignature(SigNode, vbNullString, False)

   ' If the signature is not verified, then delete the signature and set
   ' the return code to "Invalid".  Otherwise, set the return code to
   ' "Valid".

   If (Not(SigStatus = UFL_DS_OK)) Then
```

```
      TempNode.DeleteSignature SigNode
      CheckSignature = "Invalid"
   Else
      CheckSignature = "Valid"
   End If

End Function
```

DereferenceEx

Description

Use this function to locate a particular IFormNodeP, to locate a cell in a particular group, or to locate a data item in a particular datagroup. The node that this function operates on is used as the starting point of the search.

Note: It is not necessary to call this function when you are using XForms. The ReplaceXFormsInstance and ExtractXFormsInstancefunctions perform this task automatically.

Function

```
Function DereferenceEx(
   theScheme As String,
   theReference As String,
   theReferenceCode As Long,
   referenceType As Long,
   theNSNode As IFormNodeP
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theReference	**String**	The reference string.
theReferenceCode	**Long**	Reserved. This must be 0.

Expression	Type	Description
referenceType	**Long**	One of the following constants: **UFL_OPTION_REFERENCE** **UFL_ITEM_REFERENCE** **UFL_PAGE_REFERENCE** **UFL_ARRAY_REFERENCE** **UFL_GROUP_REFERENCE** **UFL_DATAGROUP_REFERENCE** If it is an option or argument reference, bitwise **OR** (|) with one of: **UFL_SEARCH** **UFL_SEARCH_AND_CREATE** If it is a group or datagroup reference, bitwise **OR** (|) with one of: **UFL_FIRST**
theNSNode	**IFormNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see the note about namespace below). Use null if the node that this function is operating on has inherited the necessary namespaces.

Returns

The IFormNodeP defined by the reference string or null if the referenced node does not exist and **UFL_SEARCH_AND_CREATE** is not specified. On error, the function throws an exception.

Notes

IFormNodeP

Before you decide which IFormNodeP to use this function on, be sure you understand the following:

1. The IFormNodeP supplied can never be more than one level in the hierarchy above the starting point of the reference string. For example, if the reference string begins with an option, then the IFormNodeP can be no higher in the hierarchy than an item.
2. If the IFormNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the function will attempt to locate a common ancestor. The function will locate the ancestor of the IFormNodeP that is one level in the hierarchy above the starting point of the reference string. The function will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the IFormNodeP and the reference string), the function will fail. For example, given a IFormNodeP that

represents *field_1* and a reference of *field_2*, the function will access the *page* node above *field_1*, and will then try to locate *field_2* below that node. If the two fields were not on the same page, the function would fail.

3. DereferenceEx does not support the XForms scheme.

Creating a Reference String

For general information about creating a reference string, see "References" on page 6.

Reference strings for groups or datagroups follow this format:

```
page.group or page.datagroup
```

In both cases, the page component is optional, and is only required if you want to search a different page than the one containing your reference node.

For example, to refer to the "State" group of cells on PAGE1 of the form, you would use:

```
PAGE1.State
```

Locating Cells or Data Items

If you want to locate a cell or a data item, you must perform a bitwise OR with UFL_FIRST or UFL_NEXT. UFL_FIRST will locate the first cell or data item in the page. UFL_NEXT will locate the next cell or data item. This allows you to loop through all the cells or data item on a page until you have found the one you want.

Note that groups and datagroups are limited to a single page, and that your search will likewise be limited to a single page.

Creating a Node

For an option or argument reference, you can have the library create a node that does not exist. To do so, perform a bitwise OR of UFL_SEARCH_AND_CREATE to the *referenceType* parameter; otherwise, perform a bitwise OR of UFL_SEARCH to the *referenceType* variable and the function will return null if the node does not exist.

Determining Namespace

In some cases, you may want to use the **DereferenceEx** function to locate a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
Label1Node.DereferenceEx(vbNullString, "Field1.processing:myValue", 0,
   UFL_OPTION_REFERENCE, vbNullString)
```

In this example, the **DereferenceEx** function would fail. The function cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as the last parameter in the function:

```
Label1Node.DereferenceEx(vbNullString, "Field1.processing:myValue", 0,
   UFL_OPTION_REFERENCE, Field1Node)
```

Example

The following example adds a label to a form. A node is passed into the function, which then uses **GetLiteralByRefEx** to read the value of a field. The function then uses **DereferenceEx** to locate the field node, and creates a label node as a sibling using **Create**. Finally, the function creates a value for the new label node using **SetLiteralByRef**.

```
Sub AddLabel(Form)

   Dim TempNode, XFDL ' objects
   Dim Name ' strings

   Set TempNode = Form

   ' Get the value of the NameField.value option node

   Name = TempNode.GetLiteralByRefEx(vbNullString, _
      "PAGE1.NameField.value", 0, vbNullString, Nothing)

   ' Locate the NameField item node in the first page of the form

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
   "PAGE1.NameField", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get an XFDL object

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")

   ' Create a label.  This label is created as a sibling of the NameField,
   ' and is named NameLabel.

   Set TempNode = XFDL.Create(TempNode, UFL_AFTER_SIBLING, "label", _
      vbNullString, vbNullString, "NameLabel")

   ' Create a value option for the label.  This option is assigned the
   ' value of Name (as read from the field)

   TempNode.SetLiteralByRefEx vbNullString, "value", 0, vbNullString, _
      Nothing, Name
End Sub
```

Destroy

Description

This function destroys the indicated IFormNodeP. All children of the specified IFormNodeP are also destroyed.

Function

Sub Destroy()

Parameters

There are no parameters for this function.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Notes

Digital Signatures

You cannot destroy a signed item, except in the case of destroying an entire signed form. Destroying a signed item breaks the digital signature, resulting in an exception.

Example

The following function uses **GetChildren** and **GetNext** to walk through an entire form and deletes all information that is not in the XFDL namespace. The function uses **IsXFDL** to determine which nodes are in the XFDL namespace. Note that the function assumes that you are passing the root node of the form on the first call. Subsequent calls occur through recursion, which may provide any level of node.

```
Sub DeleteCustomInfo(Node)

   Dim MainNode, TestNode  ' objects
   Dim TempInt

   ' Set the MainNode to be the child of the provided node.

   Set MainNode = Node.GetChildren

   ' Use recursion to step through each node in the form.  This routine
   ' walks to the last node in each page, then traverses back up the tree,
   ' deleting nodes that are not in the XFDL namespace as it goes.

   Do While (Not(MainNode Is Nothing))
      Set TestNode = MainNode.getNext
      DeleteCustomInfo(MainNode)
      Set MainNode = TestNode
   Loop

   ' Check to see if the node passed to the routine is in the XFDL
   ' namespace.  If not, delete the node.

   If Node.isXFDL = False Then
      Node.Destroy
   End If

End Sub
```

Duplicate

Description

This function makes a copy of a node. The duplicate node can be attached to any other node as either a sibling or a child, or can be stored as a separate node structure (that is, as a separate form). The new node can also be assigned a new identifier, as indicated by the *theIdentifier* parameter. All of the properties of the original node are duplicated, including any children and any namespace settings.

Note: If you duplicate a node that is not in the XFDL namespace, the namespace is copied as part of the duplicated node, but is not set globally.

Function

```
Function Duplicate(
   baseNode As IFormNodeP,
   where As Long,
   theIdentifier As String
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
baseNode	**IFormNodeP**	The **formNodeP** to attach the new copy to. If null, then *origNode* is used as the *baseNode*.
where	**Long**	A constant that describes the location in relation to the supplied *'baseNode'* in which the new node should be placed. Can be one of: **UFL_APPEND_CHILD** — adds the new node as the last child of the *'baseNode'*. **UFL_AFTER_SIBLING** — adds the new node as a sibling of the *'baseNode'*, placing it immediately after that node in the form structure. **UFL_BEFORE_SIBLING** — adds the new node as a sibling of the *'baseNode'*, placing it immediately before that node in the form structure. **UFL_ORPHAN** — copies the node to a new form structure, effectively creating a separate form.
theIdentifier	**String**	A new identifier for this node. If null, the same identifier that was on the original node is used.

Returns

The duplicate node or throws an exception if an error occurs.

Example

The following function duplicates a page in the form, including all of the items on the page. First, the function uses **DereferenceEx** to locate the PAGE2 node and then the PAGE3 node. Next, the function uses Duplicate to create a copy of the PAGE2 node and place it after the PAGE3 node.

```
Sub DuplicatePage(Form)

   Dim CopyNode, DestinationNode  ' objects

   Set CopyNode = Form
   Set DestinationNode = Form

   ' Locate the page 2 node.  This is the node that will be copied.

   Set CopyNode = CopyNode.DereferenceEx(vbNullString, "PAGE2", 0, _
      UFL_PAGE_REFERENCE, Nothing)

   ' Locate the last page node.  The copy will be inserted to the right
   ' of this node in the form structure.  In other words, the copy will
   ' become the last page.

   Set DestinationNode = DestinationNode.DereferenceEx(vbNullString, _
      "PAGE3", 0, UFL_PAGE_REFERENCE, Nothing)

   ' Duplicate the page 2 node, inserting it after the page 3 node. The
   ' new page will be named PAGE4.

   Set CopyNode = CopyNode.Duplicate(DestinationNode, UFL_AFTER_SIBLING,_
      "PAGE4")

End Sub
```

EncloseFile

Description

This function encloses a file in a form. The file must be accessible on the local computer. The IFormNodeP may refer to either a page node or an item node. If the IFormNodeP is a page node, the function creates a data item in that page to contain the enclosure. If the IFormNodeP is an item node, it must be a data item, and the function encloses the file in that node.

The file is enclosed using base64-gzip encoding.

Function

```
Function EncloseFile(
   theFile As String,
   mimeType As String,
   dataGroup As String,
   identifier As String
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
theFile	**String**	The path to the file on the local drive to enclose.
mimeType	**String**	The MIME type of the file. If null, the library will attempt to find a suitable MIME type for the file.
dataGroup	**String**	The data group to which this file should belong. If the *aNode* parameter is a page node, you must provide this parameter. If *aNode* parameter is an item node, you may use null to keep the current *datagroup* option or provide a different value to overwrite the option.

Expression	Type	Description
theIdentifier	**String**	The identifier to assign to the new data item if one is created. If null, either the current name is used or a unique name is automatically generated for the new data item.

Returns

The **IFormNodeP** of the item that contains the enclosure or throws an exception if an error occurs.

Example

The following function encloses a file in a form. The function receives a form, uses **DereferenceEx** to locate the PAGE2 node in the form, and then uses **EncloseFile** to enclose a text file in the *Notices* datagroup. The file is stored in a data item named *PersonnelNotice*.

```
Sub AttachFile(Form)
   Dim TempNode  ' object

   Set TempNode = Form

   ' Locate the PAGE2 node.

   Set TempNode = TempNode.DereferenceEx(vbNullString, "PAGE2", 0, _
      UFL_PAGE_REFERENCE, Nothing)

   ' Create an enclosure in this page.  The enclosure is stored in a data
   ' item called PersonnelNotice, is linked to the Notices datagroup, and
   ' encloses a text file from the local drive called Notice.txt.

   Set TempNode = TempNode.EncloseFile("c:\Notice.txt", "text/plain", _
      "Notices", "PersonnelNotice")

End Sub
```

EncloseInstance

Description

This function modifies one instance in the data model, either updating information or appending information. Note that the form must have an existing data model.

Call this function on the root node of the form or an XML instance node.

Note: Use caution when calling this function. It can be used to overwrite signed instance data.

Function

```
Sub EncloseInstance(
   theInstanceID As String,
   theFile As String,
   theFlags As Long,
   theScheme As String,
   theRootReference As String,
   theNSNode As IFormNodeP,
   replaceNode As Boolean)
```

Parameters

Expression	Type	Description
theInstanceID	**String**	The ID of the instance node to create or replace. This is defined by the *id* attribute of that node, and is case sensitive. If *theNode* parameter is the instance node you want to replace, set this parameter to null.
theFile	**String**	The path to the file on the local drive that contains the XML instance.
theFlags	**Long**	Reserved. Must be 0.
theScheme	**String**	Reserved. Must be null.
theRoot Reference	**String**	A reference to the node you want to replace or append children to. This reference is relative to the instance node. Use null to default to the instance node.
theNSNode	**IFormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use null if the node that this function is operating on has inherited the necessary namespaces.
replaceNode	**Boolean**	If True, the node specified by *theRootReference* is replaced with data. If False, the data is appended as the last child of *theRootReference* node.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Example

The following example shows a function that takes the root node of a form and updates the XML instance called ″data″.

```
Sub UpdateDataInstance(Form)
   Form.EncloseInstance "Test", "c:\DataInstance.txt", 0, vbNullString, _
      vbNullString, Nothing, False
End Sub
```

ExtractFile

Description

This function will extract an enclosure contained in a node and save it to a file on the local computer. Note that this function does not remove the enclosure from the form.

Function

```
Sub ExtractFile(
   theFile As String)
```

Parameters

Expression	Type	Description
theFile	**String**	The path showing where to store the file on the local drive. Any existing file will be overwritten.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Example

The following function extracts an attachment from a form and saves it to disk. First, the function uses **DereferenceEx** to locate the data item containing the attachment, then it uses **ExtractFile** to write the attachment to a file on the local drive.

```
Sub SaveAttachment(Form)

   Dim TempNode  ' object

   Set TempNode = Form

   ' Locate the PAGE3.DATA1 item, which contains the enclosure.

   Set TempNode = TempNode.DereferenceEx(vbNullString, "PAGE3.DATA1", _
      0, UFL_ITEM_REFERENCE, Nothing)

   ' Extract the enclosure from the data item and save it to disk as
   ' c:\Review1.doc

   TempNode.ExtractFile "c:\Review1.doc"

End Sub
```

ExtractInstance

Description

This function copies an instance from a form's XML model to a file. Note that this function does not remove the instance from the form.

Call this function on the root node of the form or an XML instance node.

Function

```
Sub ExtractInstance(
   theInstanceID As String,
   theFilter As IFormNodeP,
   includeNamespaces As String,
   theFile As String,
   theFlags As Long,
   theScheme As String,
   theRootReference As String,
   theNSNode As IFormNodeP)
```

Parameters

Expression	Type	Description
theInstanceID	**String**	The ID of the instance node to extract. This is defined by the *id* attribute of that node. If *theNode* parameter is the instance node you want to extract, set this parameter to .
theFilter	**IFormNodeP**	An item in the form, such as a button or cell, that defines the filtering for the instance. Filtering of elements is controlled by the transmit filters in the item. If all of an element's bound options are filtered out, then the element is also filtered out. Use for no filtering.
included Namespaces	**String**	If set to null, a definition for each inherited namespace is added to the root node of the instance when it is extracted. To filter the namespaces, list the prefixes for those namespaces you want to include in the instance, separated by spaces. For example, to include only the *xfdl* and *custom* namespaces, you would set this parameter to: `xfdl custom` Use *#default* to indicate the default namespace for the instance. Use an empty string ("") to include only those namespaces that are used by the instance. Namespaces that are used in the instance are always included, regardless of this setting.
theFile	**String**	The path to the file on the local drive that will contain the XML instance.
theFlags	**Long**	Reserved. This must be 0.
theScheme	**String**	Reserved. Must be null.
theRootReference	**String**	A reference to the root node you want to extract. This reference is relative to the instance node. Use null to default to the instance node.
theNSNode	**IFormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use null if the node that this function is operating on has inherited the necessary namespaces.

Returns

Nothing if call is successful or throws a generic exception (**UWIException**) if an error occurs.

Example

The following example shows a function that takes the root node of a form and extracts an XML instance.

```
Sub SaveDataInstance(Form)

   Form.ExtractInstance "Test", Nothing, vbNullString, _
      "c:\InstanceData.txt", 0, vbNullString, vbNullString, Nothing

End Sub
```

ExtractXFormsInstance

Description

This function copies an XForms instance to a file or a memory block. This function does not remove the instance from the form.

Call this function on the root node of the form or an instance node.

Note: This function automatically updates the XForms data model.

Function

```
Sub ExtractXformsInstance(
   theModelID As String,
   theNodeRef As String,
   writeRelevant As Boolean,
   IgnoreFailures As Boolean,
   theNSNode As IFormNodeP,
   theFilename As String,
   theMemoryBlock As Variant)
```

Parameters

Expression	Type	Description
theModelID	**String**	The ID of the model to extract. Use vbnullstring to extract the default model.
theNodeRef	**String**	An XPath reference to a node in the instance. This node and all of its children are copied. Leave blank to extract the entire instance.
writeRelevant	**Boolean**	If True, writes only relevant instance data.
ignoreFailures	**Boolean**	If True, ignores constraint or validation failures.
theNSNode	**IFormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use nothing if the node that this function is operating on has inherited the necessary namespaces.
theFilename	**String**	The name and path of the file to write to. Use vbnullstring to write to the output memory block.
theMemoryBlock	**Variant**	The memory block that represents the instance if you are not writing to a file.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Example

The following example shows a routine that takes the root node of a form, extracts an XForms instance, and writes it to a file called ″InstanceData.xml″.

```
Sub SaveDataInstance(Form)

   Form.ExtractXFormsInstance"model1",
      "instance('instance1')loanrecord.user_personal_info", True, False,
       Nothing, "c:\InstanceData.xml", Nothing

End Sub
```

GetAttribute

Description

This function returns the value of a specific attribute for a node. For example, the following XFDL represents a MIME data node:

```
<mimedata encoding="base64"></mimedata>
```

In this sample, you could use **GetAttribute** to obtain the value of the encoding attribute, which would be ″base64″.

Function

```
Function GetAttribute(
   theNamespaceURI As String,
   theAttribute As String
   ) As String
```

Parameters

Expression	Type	Description
theNamespaceURI	**String**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**String**	The local name of the attribute. For example, *encoding*.

Returns

The attribute's value or throws an exception if an error occurs. If the attribute is empty or does not exist, the function returns **null**.

Notes

Namespaces

If you refer to an attribute with a namespace prefix, *getAttribute* first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Special Attributes

Forms generally use three special attributes that are not in an explicitly defined namespace and which require special commands to retrieve.

The first is the default namespace attribute, which looks like this:

```
xmlns="http://www.ibm.com/xmlns/prod/XFDL/7.0"
```

To retrieve this attribute, you must use a namespace URI of null and the attribute name *xmlns*.

The second special attribute is a namespace declaration, which looks like this:

```
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

To retrieve this sort of attribute, you must use the namespace URI *http://www.w3.org/2000/xmlns* and the appropriate attribute name, such as *custom*.

Finally, there is the language attribute, which looks like this:

```
xml:lang="en-GB"
```

To retrieve this sort of attribute, you must use the namespace URI *http://www.w3.org/XML/1998/namespace* and the attribute name *lang*.

Example

The following function uses **DereferenceEx** to locate a node that contains an attachment. It then calls **GetAttribute** to retrieve the encoding type used, and returns that value.

```
Function GetEncodingType(Form)

   Dim TempNode  ' object

   Set TempNode = Form

   ' Locate the mimedata option for the DATA1 node in PAGE3, which
   ' contains an attachment.

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
      "PAGE3.DATA1.mimedata", 0, UFL_OPTION_REFERENCE, Nothing)

   ' Get the encoding type of the data node.

   GetEncodingType = TempNode.GetAttribute( _
      "http://www.ibm.com/xmlns/prod/XFDL/7.0", "encoding")

End Function
```

GetCertificateList

Description

This function locates all available certificates that can be used by a particular signature button. The certificates are filtered according to the signature engine defined in the *signformat* option of the button, and according to the filters defined in the *signdetails* option of the button.

This function returns the valid certificates in an undetermined order. This means that you cannot rely on the certificates being listed in the same order each time you call this function.

Function

```
Function GetCertificateList(
   theFilters As String,
   theStatus As Long
   ) As CertificateList
```

Parameters

Expression	Type	Description
theFilters	**String**	A string that is used to filter the subject attribute of the certificate. If the subject attribute include this substring, then that certificate will be listed. For example, you might filter against a name, such as "John Doe", or an e-mail address, such as "jdoe@ibm.com". Note that this filter is in addition to the other filters defined in the *signdetails* option of the button. If null is passed, then only the filters in the *signdetails* option are used.
theStatus	**Long**	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

An array containing the list of certificates objects.

Example

The following function uses **DereferenceEx** to locate a signature button in the form. It then calls **GetCertificateList** to get a list of valid certificates for that button. The function then loops through the available certificates, using

GetDataByPath to check the common name of each certicate. When it finds the certificate with the common name of ″TJones″, it calls **SignForm** and uses that certificate to sign the form.

```
Sub ApplySignature(Form)

   Dim SigNode, SigObject  ' objects
   Dim TheCerts  ' CertificateList
   Dim CommonName  ' String
   Dim Cert  ' ICertificate

   ' Get the SignatureButton node

   Set SigNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.SignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get available certificates for that button

   Set TheCerts = SigNode.GetCertificateList(vbNullString, 1) 'vbNull

   ' Test each of the available certificates to see if it has a common
   ' name of "TJones".  If it does, use that certificate to sign
   ' the form.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "TJones" Then
         Set SigObject = SigNode.SignForm(TheCerts(1), Nothing, 1)
            ' vbNull
      End If
   Next

End Sub
```

GetChildren

Description

This function, along with **GetParent**, is used to traverse vertically along the form hierarchy. **GetChildren** returns the first child of the indicated node. If the node has no children, null is returned. All children of a particular IFormNodeP can be traversed using an iterator, such as a while loop, in combination with **GetNext**.

Function

```
Function GetChildren() As IFormNodeP
```

Parameters

There are no parameters for this function.

Returns

The that represents the child or if no such child exists.

Example

getChildren returns the first child node of *PAGE1.NAMELABEL* that is *PAGE1.NAMELABEL.value*.

The following sample function receives any form node and returns a reference to the last page of the form. First, the function uses **GetNodeType** to determine if the supplied node is a the form level. If so, the function uses **GetChildren** to locate the first page node. If not, the function uses **GetParent** to locate a page level node. The function then uses **GetNext** to locate the last page in the form and calls **GetReferenceEx** to retrieve a reference to that node.

```
Function GetLastPage(Node)

   Dim MainNode, TestNode ' objects

   Set MainNode = Node

   ' Locate the page level node that is the nearest ancestor or child of
   ' the current node.  If the node is a form node, get the child.  If the
   ' node is any other type, get the parent until a page node is
   ' retrieved.

   If MainNode.getNodeType = UFL_FORM Then
      Set MainNode = MainNode.GetChildren
   Else

      ' Locate the page node of the form by iterating through the parents
      ' of the supplied node.  At the end of this loop, MainNode will be
      ' a page node.

      Do While (Not(MainNode.GetNodeType = UFL_PAGE))
         Set MainNode = MainNode.GetParent
      Loop
   End If

   ' Set TestNode to be the MainNode (the page node) before beginning the
   ' next loop.

   Set TestNode = MainNode

   ' Locate the last page node by iterating through the siblings of the
   ' global page node.  At the end of this loop, TestNode will be Nothing
   ' and MainNode will be the last page node.

   Do While (Not(TestNode Is Nothing))
      Set MainNode = TestNode
      Set TestNode = TestNode.GetNext
   Loop

   ' Get a reference to the MainNode and return that reference to the
   ' caller.

   GetLastPage = MainNode.GetReferenceEx(vbNullString, Nothing, Nothing,_
       False)

End Function
```

GetFormVersion

Description

This function determines the XFDL version of a form. You can call this function on any form node that is in the XFDL namespace.

Function

```
Function GetFormVersion() As Integer
```

Parameters

There are no parameters for this function.

Returns

An integer in the form of &HMMmm0300, where MM is the major number and mm is the minor number. For example, a version 6.3 form would return: &H06030300.

Example

The following function accepts a form node and returns a boolean that indicates whether the form is version 6.5 or higher.

```
Function CheckVersion(TheNode)

   If (TheNode.GetFormVersion >= &H06050300) Then
      CheckVersion = True
   Else
      CheckVersion = False
   End If

End Function
```

GetIdentifier

Description

This function retrieves the identifier of a node. This is either the scope identifier or option name for the node.

Function

Function GetIdentifier() As **String**

Parameters

There are no parameters for this function.

Returns

A string containing the identifier of the node or throws an exception if an error occurs. If the identifier is empty or does not exist, the function returns **null**.

Example

The following function locates the first value option in a single page form, and assumes that you pass it the root node of the form. First, the function uses **GetChildren**and **GetNext** to locate the first item in the first page. The function then uses nested loops and **GetNext** to iterate through each option for each item in the form, testing each option with **GetIdentifier** until it locates a value node. Finally, the function returns the value node, or a **null** node if no value node was found.

```
Function LocateFirstValue(Form)

   Dim ItemNode, OptionNode  ' objects
   Dim Found  ' Boolean
```

```
    Set ItemNode = Form
    Found = False

    ' Find the first item on the first page of the form. This takes three
    ' separate calls, since we do not know the name of the first page or
    ' item.

    Set ItemNode = ItemNode.GetChildren
    Set ItemNode = ItemNode.GetNext
    Set ItemNode = ItemNode.GetChildren

    ' Iterate through the items until one is found that contains a value
    ' option or a null node is reached.

    Do While (Not(ItemNode is Nothing))

       ' Find the first option in the current item.

       Set OptionNode = ItemNode.GetChildren

       ' Iterate through the options until a "value" option is found or a
       ' null node is reached.

       Do While (Not(OptionNode is Nothing))
          If OptionNode.GetIdentifier= "value" Then
             Found = True
             Exit Do
          End If
          Set OptionNode = OptionNode.GetNext
       Loop

       If Found Then Exit Do
       Set ItemNode = ItemNode.GetNext

    Loop

    Set LocateFirstValue = OptionNode

End Function
```

GetLiteralByRefEx

Description

This function finds a particular IFormNodeP on the basis of a reference string. The node you call this function on is used as the starting point for the search unless you provide an absolute reference. Once the IFormNodeP is found, its literal is retrieved.

Note: It is not necessary to call this function when you are using XForms. The ReplaceXFormsInstance and ExtractXFormsInstancefunctions perform this task automatically.

Function

```
Function GetLiteralByRefEx(
   theScheme As String,
   theReference As String,
   theReferenceCode As Long,
   theCharSet As String,
   theNSNode As IFormNodeP
   ) As String
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theReference	**String**	The reference string.
theReferenceCode	**Long**	Reserved. This must be 0.
theCharSet	**String**	The character set you want to use to view the literal string. Use null for ANSI/Unicode. Use **Symbol** for Symbol.
theNSNode	**IFormNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see the note about namespace below). Use null if the node that this function is operating on has inherited the necessary namespaces.

Returns

The literal string or throws an exception if an error occurs. If the literal is empty or does not exist, the function returns **null**.

Notes

IFormNodeP

Before you decide which IFormNodeP to call the function on, be sure you understand the following:

1. The IFormNodeP supplied can never be more than one level in the hierarchy above the starting point of the reference string. For example, if the reference string begins with an option, then the IFormNodeP can be no higher in the hierarchy than an item.
2. If the IFormNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the function will attempt to locate a common ancestor. The function will locate the ancestor of the IFormNodeP that is one level in the hierarchy above the starting point of the reference string. The function will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the IFormNodeP and the reference string), the function will fail.

   For example, given a IFormNodeP that represents ″field_1″ and a reference of ″field_2″, the function will access the ″page″ node above ″field_1″, and will then try to locate ″field_2″ below that node. If the two fields are not on the same page, the function will fail.
3. If the IFormNodeP is at the argument level, the search will not start from that point. Instead, the nearest ancestor that is at the option level will be used as the starting point for the search.

Creating a Reference String

For more information about creating a reference, see “References” on page 6.

Determining Namespace

In some cases, you may want to use the **GetLiteralByRefEx** function to get the literal of a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
Label1Node.GetLiteralByRefEx(vbNullString, "Field1.processing:myValue",
   0, vbNullString, vbNullString)
```

In this example, the **GetLiteralByRefEx** function would fail. The function cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as a parameter in the function:

```
Label1Node.GetLiteralByRefEx(vbNullString, "Field1.processing:myValue",
   0, vbNullString, Field1Node)
```

Example

The following example adds a label to a form. A node is passed into the function, which then uses **GetLiteralByRefEx** to read the value of a field. The function then uses **DereferenceEx** to locate the field node, and creates a label node as a sibling using **Create**. Finally, the function creates a value for the new label node using **SetLiteralByRef**.

```
Sub AddLabel(Form)

   Dim TempNode, XFDL ' objects
   Dim Name ' strings
   Set TempNode = Form

   ' Get the value of the NameField.value option node

   Name = TempNode.GetLiteralByRefEx(vbNullString, _
      "PAGE1.NameField.value", 0, vbNullString, Nothing)

   ' Locate the NameField item node in the first page of the form

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
   "PAGE1.NameField", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get an XFDL object

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")

   ' Create a label.  This label is created as a sibling of the NameField,
   ' and is named NameLabel.
```

```
    Set TempNode = XFDL.Create(TempNode, UFL_AFTER_SIBLING, "label", _
       vbNullString, vbNullString, "NameLabel")

    ' Create a value option for the label.  This option is assigned the
    ' value of Name (as read from the field)

    TempNode.SetLiteralByRefEx vbNullString, "value", 0, vbNullString, _
       Nothing, Name

End Sub
```

GetLiteralEx

Description

This function retrieves the literal of a node. The literal is returned in the specified character set.

Note: It is not necessary to call this function when you are using XForms. The ReplaceXFormsInstance and ExtractXFormsInstancefunctions perform this task automatically.

Function

```
Function GetLiteralEx(
   theCharSet As String
   ) As String
```

Parameters

Expression	Type	Description
theCharSet	**String**	The character set you want to use to view the literal string. . Use **null** for ANSI/Unicode. Use **Symbol** for Symbol.

Returns

A string containing the literal of the node or throws an exception if an error occurs. If the literal is empty or does not exist, the function returns **null**.

Example

The following function copies the value from one field to another. First, the function uses **DeferenceEx** to locate the value node of a field on page one and reads the value using **GetLiteral**. Next, teh function locates a value node on page two and writes the literal to that field using **SetLiteral**.

```
Sub CopyValue(Form)

    Dim TempNode  ' object
    Dim theName  ' string

    Set TempNode = Form

    ' Locate the NameField on page 1

    Set TempNode = TempNode.DereferenceEx(vbNullString, _
       "PAGE1.NameField.value", 0, UFL_OPTION_REFERENCE, Nothing)

    ' Get the literal from the value node
```

```
    theName = TempNode.GetLiteralEx

    ' Locate the NameField on page 2

    Set TempNode = TempNode.DereferenceEx(vbNullString, _
       "PAGE2.NameField.value", 0, UFL_OPTION_REFERENCE, Nothing)

    ' Write the literal that was read from the first value node

    TempNode.SetLiteral theName

End Sub
```

GetLocalName

Description

This function returns the *local name* of a given node. The local name is determined by the XML tag that represents that node. For example, examine the following XML fragment:

```
<page sid="PAGE1">
   <global sid="global"></global>
   <field sid="testField">
      <value>Hello</value>
      <bgcolor>
         <ae>120</ae>
         <ae>120</ae>
         <ae>120</ae>
      <bgcolor>
   </field>
</page>
```

In this sample, the name of the page node is ″page″, the name of the field node is ″field″, the name of the value node is ″value″, and the name of the bgcolor node is ″bgcolor″. The bgcolor node is also the parent of three array element nodes, all of which are named ″ae″.

Note that the local name does not include any namespace prefix that might exist. For example, you might have a custom option in a different namespace as shown:

```
<field sid="testField">
   <custom:my_option>value</custom:my_option>
</field>
```

In this case, the local name of the custom option is returned without the prefix, resulting in ″my_option″.

Function

```
Function GetLocalName() As String
```

Parameters

There are no parameters for this function.

Returns

The name of the node or throws an exception if an error occurs.

Example

The following function uses **GetChildren** and **GetNext** to walk through the entire form recursively. While walking through the form, the function uses **IsXFDL** and **GetLocalName** to test each node in the form and determine whether it is an XFDL label node. If so, the function uses **SetLiteralByRefEx** to set the label's background color to green.

```
Sub ChangeLabelColor(Node)

   Dim TempNode, BGColorNode  ' objects

   ' Use recursion to step through each node in the form

   Set TempNode = Node.GetChildren

   Do While (Not(TempNode Is Nothing))
      ChangeLabelColor(TempNode)
      Set TempNode = TempNode.GetNext
   Loop

   ' If the node is a label in the XFDL namespace, locate the background
   ' color child node and change its value to green.

   If ((Node.IsXFDL) And (Node.GetLocalName = "label")) Then
      Node.SetLiteralByRefEx vbNullString, "bgcolor[0]", 0, _
         vbNullString, Nothing, "green"
   End If

End Sub
```

GetNamespaceURI

Description

This function returns the *namespace URI* for the node.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
Function GetNamespaceURI() As String
```

Parameters

There are no parameters for this function.

Returns

The namespace URI or throws an exception if an error occurs.

Example

The following function uses **GetChildren** and **GetNext** to walk through an entire form. While walking through the form, it uses **GetNamespaceURI** to determine whether each node is in the *Custom* namespace identified by the following URI: *http://www.ibm.com/xmlns/prod/XFDL/Custom*. If so, the function destroys the node. This function assumes that you are passing the root node of the form on the first call. Subsequent calls occur through recursion, which may provide any level of node.

```
Sub DeleteCustomNamespace(Node)

   Dim MainNode, TestNode  ' objects
   Dim TempInt

   ' Set the MainNode to be the child of the provided node.

   Set MainNode = Node.GetChildren

   ' Use recursion to step through each node in the form.  This routine
   ' walks to the last node in each page, then traverses back up the tree,
   ' deleting nodes that are not in the XFDL namespace as it goes.

   Do While (Not(MainNode Is Nothing))
      Set TestNode = MainNode.GetNext
      DeleteCustomInfo(MainNode)
      Set MainNode = TestNode
   Loop

   ' Check to see if the node passed to the routine is in the Custom
   ' namespace.  If so, delete the node.

   If Node.GetNamespaceURI = "http://www.ibm.com/xmlns/prod/XFDL/Custom" Then
      Node.Destroy
   End If

End Sub
```

GetNamespaceURIFromPrefix

Description

This function returns the *namespace URI* that corresponds to a specific prefix. You can call this function from any node in the form, as long as that node either declares or inherits the namespace in question.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
Function GetNamespaceURIFromPrefix(
   thePrefix As String
   ) As String
```

Parameters

Expression	Type	Description
thePrefix	**String**	The namespace prefix. For example, *xfdl*.

Returns

The namespace URI or throws an exception if an error occurs. If the namespace URI is not declared, the result is **null**.

Example

The following function copies an option in the custom namespace from one form to another. First, it uses **GetNamespaceURIFromPrefix** to get the URI for the custom namespace from Form1, then uses **AddNamespace** to add that namespace to Form2. Next, it uses **DereferenceEx** to locate the custom node in Form1 and the global page's global item in Form2. Finally, it uses **Duplicate** to copy the custom node from Form1 into Form2, creating it as a child of the global item.

```
Sub CopyCustomInfo(Form1, Form2)

   Dim TheURI  ' String
   Dim TempNode, DuplicateNode, GlobalNode  ' IFormNodeP

   ' Get the URI for the custom namespace in Form1.

   TheURI = Form1.GetNamespaceURIFromPrefix("custom")

   ' Create the custom namespace in Form2 using the URI from Form1.

   Form2.AddNamespace TheURI, "custom"

   ' Locate the custom processing node in Form1.

   Set TempNode = Form1.DereferenceEx(vbNullString, _
      "global.global.custom:processing", 0, UFL_OPTION_REFERENCE, _
      Nothing)

   ' Locate the global item in Form2

   Set GlobalNode = Form2.DereferenceEx(vbNullString, "global.global", _
      0, UFL_ITEM_REFERENCE, Nothing)

   ' Copy the custom node from Form1 and insert it as the child of the
   ' global item in Form2.

   Set DuplicateNode = TempNode.duplicate(GlobalNode, _
      UFL_APPEND_CHILD, vbNullString)

End Sub
```

GetNext

Description

This function, along with **GetPrevious**, is used to traverse horizontally along the form hierarchy. **GetNext** returns the next node in the tree. For instance, the page node corresponding to the first page of your form can be reached by calling **GetNext** on the global page node.

Function

```
Function GetNext() As IFormNodeP
```

Parameters

There are no parameters for this function.

Returns

The IFormNodeP that represents the next node or null if no such node exists.An exception is thrown if an error occurs.

Example

The following sample function receives any form node and returns a reference to the last page of the form. First, the function uses **GetNodeType** to determine if the supplied node is a the form level. If so, the function uses **GetChildren** to locate the first page node. If not, the function uses **GetParent** to locate a page level node. The function then uses **GetNext** to locate the last page in the form and calls **GetReferenceEx** to retrieve a reference to that node.

```
Function GetLastPage(Node)

   Dim MainNode, TestNode ' objects

   Set MainNode = Node

   ' Locate the page level node that is the nearest ancestor or child of
   ' the current node.  If the node is a form node, get the child.  If the
   ' node is any other type, get the parent until a page node is
   ' retrieved.

   If MainNode.getNodeType = UFL_FORM Then
      Set MainNode = MainNode.GetChildren
   Else

      ' Locate the page node of the form by iterating through the parents
      ' of the supplied node.  At the end of this loop, MainNode will be
      ' a page node.

      Do While (Not(MainNode.GetNodeType = UFL_PAGE))
         Set MainNode = MainNode.GetParent
      Loop
   End If

   ' Set TestNode to be the MainNode (the page node) before beginning the
   ' next loop.

   Set TestNode = MainNode

   ' Locate the last page node by iterating through the siblings of the
   ' global page node.  At the end of this loop, TestNode will be Nothing
   ' and MainNode will be the last page node.
```

```
   Do While (Not(TestNode Is Nothing))
      Set MainNode = TestNode
      Set TestNode = TestNode.GetNext
   Loop

   ' Get a reference to the MainNode and return that reference to the
   ' caller.

   GetLastPage = MainNode.GetReferenceEx(vbNullString, Nothing, Nothing,_
       False)

End Function
```

GetNodeType

Description

This function returns the type for a node (for example, page, item, option, and so on). This allows you to quickly determine the type of node you are working with and what depth you are at in the node hierarchy.

Function

```
Function GetNodeType() As Long
```

Parameters

There are no parameters for this function.

Returns

This method throws a generic exception (**UWIException**) if an error occurs.

One of the following types:

- UFL_FORM — The root node of the form.
- UFL_PAGE — A page level node.
- UFL_ITEM — An item level node.
- UFL_OPTION — An option level node.
- UFL_ARRAY — An argument level node, such as an array element.

This function throws an exception if an error occurs.

Example

The following function recieves a node below the page level and uses **GetParent** to ascend the hierarchy until it reaches a page node, as detected by **GetNodeType**.

The following sample function receives any form node and returns a reference to the last page of the form. First, the function uses **GetNodeType** to determine if the supplied node is a the form level. If so, the function uses **GetChildren** to locate the first page node. If not, the function uses **GetParent** to locate a page level node. The function then uses **GetNext** to locate the last page in the form and calls **GetReferenceEx** to retrieve a reference to that node.

```
Function GetLastPage(Node)
   Dim MainNode, TestNode ' objects

   Set MainNode = Node
```

```
    ' Locate the page level node that is the nearest ancestor or child of
    ' the current node.  If the node is a form node, get the child.  If the
    ' node is any other type, get the parent until a page node is
    ' retrieved.

    If MainNode.getNodeType = UFL_FORM Then
       Set MainNode = MainNode.GetChildren
    Else
       ' Locate the page node of the form by iterating through the parents
       ' of the supplied node.  At the end of this loop, MainNode will be
       ' a page node.

       Do While (Not(MainNode.GetNodeType = UFL_PAGE))
          Set MainNode = MainNode.GetParent
       Loop
    End If

    ' Set TestNode to be the MainNode (the page node) before beginning the
    ' next loop.

    Set TestNode = MainNode

    ' Locate the last page node by iterating through the siblings of the
    ' global page node.  At the end of this loop, TestNode will be Nothing
    ' and MainNode will be the last page node.

    Do While (Not(TestNode Is Nothing))
       Set MainNode = TestNode
       Set TestNode = TestNode.GetNext
    Loop

    ' Get a reference to the MainNode and return that reference to the
    ' caller.

    GetLastPage = MainNode.GetReferenceEx(vbNullString, Nothing, Nothing,_
        False)

End Function
```

GetParent

Description

This function, along with **GetChildren**, is used to traverse vertically along the form hierarchy. **GetParent** returns the parent of a node. If the node has no parent, null is returned. A form's structure can be traversed up to the root node using an iterator such as a while loop.

Function

```
Function GetParent() As IFormNodeP
```

Parameters

There are no parameters for this function.

Returns

The IFormNodeP that represents the parent node or null if no such parent exists. If an error occurs, an exception is thrown.

Example

getParent returns the parent node of *PAGE1.AGEFIELD.size*, that is, *PAGE1.AGEFIELD*.

The following sample function receives any form node and returns a reference to the last page of the form. First, the function uses **GetNodeType** to determine if the supplied node is a the form level. If so, the function uses **GetChildren** to locate the first page node. If not, the function uses **GetParent** to locate a page level node. The function then uses **GetNext** to locate the last page in the form and calls **GetReferenceEx** to retrieve a reference to that node.

```
Function GetLastPage(Node)

   Dim MainNode, TestNode ' objects

   Set MainNode = Node

   ' Locate the page level node that is the nearest ancestor or child of
   ' the current node.  If the node is a form node, get the child.  If the
   ' node is any other type, get the parent until a page node is
   ' retrieved.

   If MainNode.GetNodeType = UFL_FORM Then
      Set MainNode = MainNode.GetChildren
   Else
      ' Locate the page node of the form by iterating through the parents
      ' of the supplied node.  At the end of this loop, MainNode will be
      ' a page node.

      Do While (Not(MainNode.GetNodeType = UFL_PAGE))
         Set MainNode = MainNode.GetParent
      Loop
   End If

   ' Set TestNode to be the MainNode (the page node) before beginning the
   ' next loop.

   Set TestNode = MainNode

   ' Locate the last page node by iterating through the siblings of the
   ' global page node.  At the end of this loop, TestNode will be Nothing
   ' and MainNode will be the last page node.

   Do While (Not(TestNode Is Nothing))
      Set MainNode = TestNode
      Set TestNode = TestNode.GetNext
   Loop

   ' Get a reference to the MainNode and return that reference to the
   ' caller.

   GetLastPage = MainNode.GetReferenceEx(vbNullString, Nothing, Nothing,_
       False)

End Function
```

GetPrefix

Description

This function returns the namespace *prefix* for the node.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Note: A given prefix may not always resolve to the same namespace. Different portions of the form may define the prefix differently. For example, the custom prefix may resolve to a different namespace on the first page of a form than it does on the following pages.

Function

```
Function GetPrefix() As String
```

Parameters

There are no parameters for this function.

Returns

The prefix for the node's namespace or throws an exception if an error occurs.

Example

The following function removes all nodes from the form that have a namespace prefix of "custom". The function walks through the form using **GetChildren** and **GetNext** in a recursive loop. While walking the form, it uses **GetPrefix** to locate nodes in the custom namespace and deletes them using **Destroy**. This function assumes that you are passing it the root node of the form.

```
Sub DeleteCustomPrefix(Node)

   Dim TempNode, TempNode2  ' IFormNodeP

   ' Use recursion to step through each node of the form.

   Set TempNode = Node.GetChildren

   Do While (Not(TempNode is Nothing))
      Set TempNode2 = TempNode.GetNext
      DeleteCustomPrefix(TempNode)
      Set TempNode = TempNode2
   Loop

   ' If the node has a namespace prefix of "custom", delete it.

   If (Node.GetPrefix = "custom") Then
      Node.Destroy
   End If

End Sub
```

GetPrefixFromNamespaceURI

Description

This function returns the namespace *prefix* for a specific namespace URIYou can call this function from any node in the form, as long as that node either declares or inherits the namespace in question.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
Function GetPrefixFromNamespaceURI(
   theURI As String
   ) As String
```

Parameters

Expression	Type	Description
theURI	**String**	The namespace URI. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`

Returns

The namespace prefix or throws an exception if an error occurs. If the namespace URI is not declared, the result is **null**.

Example

The following function adds some custom user information to the form, but assumes that the prefix used for the custom namespace is unknown. The function first uses **GetPrefixFromNamespaceURI** to determine which prefix is used for the custom namespace. It then concatenates that prefix with the tag ″:User″ to create the name for a new node. Next, it uses **DereferenceEx** to locate the global item in the global page. Finally, it uses **Create** to add a custom option called ″User″ to the form.

```
Sub AddUserInfo(Form, UserName)

   Dim XFDL, TempNode  ' objects
   Dim ThePrefix, NewName  ' Strings

   Set TempNode = Form

   ' Retrieve the prefix for the custom namespace
```

```
    ThePrefix = TempNode.GetPrefixFromNamespaceURI( _
       "http://www.ibm.com/xmlns/prod/XFDL/Custom")

    ' Create a name for a new node by concatenating the prefix with ":User"
    NewName = ThePrefix & ":User"
    ' Locate the global item in the global page so we can add a global
    ' option

    Set TempNode = TempNode.DereferenceEx(vbNullString, "global.global", _
       0, UFL_ITEM_REFERENCE, Nothing)

    ' Create an XFDL object so we can use the create function

    Set XFDL = CreateObject("PureEdge.xfdl_XFDL")

    ' Create new node in the custom namespace that represents the user of
    ' the form and give it a value of "TJones"

    Set TempNode = XFDL.Create(TempNode, UFL_APPEND_CHILD, vbNullString, _
       UserName , vbNullString, NewName)

End Sub
```

GetPrevious

Description

This function, along with **GetNext**, is used to traverse horizontally along the form hierarchy. **GetPrevious** returns the previous node in the tree. For instance, if you call **GetPrevious** one the Page1 node in your form, it will return the global page node.

Function

```
Function GetPrevious() As IFormNodeP
```

Parameters

There are no parameters for this function.

Returns

The IFormNodeP that represents the previous node or null if no such node exists. An exception is thrown if an error occurs.

Example

The following sample function receives any form node and returns a reference to the first page of the form. First, the function uses **GetNodeType** to determine if the supplied node is a the form level. If so, the function uses **GetChildren** to locate the first page node. If not, the function uses **GetParent** to locate a page level node. The function then uses **GetPrevious** to locate the first page in the form and calls **GetReferenceEx** to retrieve a reference to that node.

```
Function GetFirstPage(Node)

    Dim MainNode, TestNode  'objects

    Set MainNode = Node

    ' Locate the page level node that is the nearest ancestor or child of
```

```
    ' the current node.  If the node is a form node, get the child.  If the
    ' node is any other type, get the parent until a page node is
    ' retrieved.

    If MainNode.GetNodeType = UFL_FORM Then
       Set MainNode = MainNode.GetChildren
    Else
       ' Locate the page node of the form by iterating through the parents
       ' of the supplied node.  At the end of this loop, MainNode will be
       ' the page node.

       Do While (Not(MainNode.GetNodeType = UFL_PAGE))
          Set MainNode = MainNode.GetParent
       Loop
    End If

    ' Set TestNode to be the MainNode before beginning the next loop

    Set TestNode = MainNode

    ' Locate the first page node (the global page) by iterating through the
    ' siblings of the located page node.  At the end of this loop, TestNode
    ' will be Nothing and MainNode will be the global page node.

    Do While (Not(TestNode Is Nothing))
       Set MainNode = TestNode
       Set TestNode = TestNode.GetPrevious
    Loop

    ' Locate the first page of the form by getting the first sibling of the
    ' global page node.

    Set MainNode = MainNode.GetNext

    ' Get a reference to the page and return the reference to the caller

    GetFirstPage = MainNode.GetReferenceEx(vbNullString, Nothing, _
       Nothing, False)

End Function
```

GetReferenceEx

Description

This function returns the reference string that identifies the node. For example, a value node might return a reference of *Page1.Field1.value*. The reference will either begin at the page level of the form or at a level specified by the caller.

Function

```
Function GetReferenceEx(
   theScheme As String,
   theNSNode As IFormNodeP,
   theStartPoint As IFormNodeP,
   addNamespaces As Boolean
   ) As String
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.

Expression	Type	Description
theNSNode	**IFormNodeP**	A node that defines which namespace prefixes are used when constructing the reference. Only namespace prefixes that this node inherits are used. Use null if the node that this function is operating on has inherited the necessary namespaces.
theStartPoint	**IFormNodeP**	A node that determines the starting point of the reference. This node must be a parent of the *aNode* parameter. The reference will begin one level below the start point node. For example, if you provide a page node the reference will begin at the item level. Use null to start the reference at the page level.
addNamespaces	**Boolean**	Use True to add declarations for unknown namespaces to the namespace node (*theNSNode*). Otherwise, use False.

Returns

A string containing a reference to the node, or throws an exception if an error occurs.

Notes

Creating a Reference String

For more information about creating a reference, see "References" on page 6.

Working with Namespace Prefixes

In some cases, you may want to use the **GetReferenceEx** function to get the reference to a node that uses a different prefix for a known namespace. For example, consider the following form:

```
<label sid="Label1" xmlns:data="URI">
   <value></value>
</label>
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, *processing* and *data* are prefixes for the same namespace, since they both refer to the same URI. However, both namespaces have limited scope since they are declared at the item level. This means that *Label1* node does not understand the *processing* prefix, and that the *Field1* node does not understand the *data* prefix.

This becomes a problem if you want to refer to a namespace from a location that does not understand that namespace. For example, suppose you wanted to set the value of *Label1* to be a reference to the *myValue* node in *Field1*. Normally, you would locate the *myValue* node and use **getReferenceEx** as shown:

```
myValueNode.GetReferenceEx(vbNullString, Nothing, Nothing, False)
```

In this case, **GetReferenceEx** would return the following reference: *Page1.Field1.processing:myValue*. However, because the *processing* namespace is not

defined for *Label1*, a reference to the *processing* namespace is not understood. This means that you cannot set the value of *Label1* to equal this reference, since the node would not understand that content.

Instead, you must generate a reference that includes a known namespace prefix, such as the *data* namespace. You can do this by including a second node in the **GetReferenceEx** function. The second node must understand the appropriate namespace. For example, you could include the *Label1* node in the function, as shown:

```
myValueNode.GetReferenceEx(vbNullString, Label1Node, Nothing, False)
```

In this case, the function will substitute the *data* prefix for the *processing* prefix, since they both resolve to the same namespace. As a result, the function will return: *Page1.Field1.data:myValue*. Since the *data* prefix is defined within *Label1*, you can use this reference to set *Label1*'s value node.

Working with Unknown Namespaces

In some cases, you may want to use the **GetReferenceEx** function to get the reference to a node that uses an unknown namespace. For example, consider the following form:

```
<page sid="Page1" xmlns:processing="URI1">
   <global sid="global">
      <processing:info></processing:info>
   </global>
   <field sid="Field1" xmlns:data="URI2">
      <value></value>
      <data:info>data</data:info>
   </field>
```

In this example, you might want to store a reference to the <data:info> element in the <processing:info> element. **GetReferenceEx** would return the following reference for the <data:info> element: *Page1.Field1.data:info*. However, this reference includes the *data* namespace, which is not defined for the page global. This means that you could not store this reference in the <processing:info> element, because it would not understand the reference.

To solve this problem, you can use the *addNamespaces* flag in the **GetReferenceEx** function. When this flag is set to True, the function will add unknown namespaces to the *theNSNode*.

For example, if you set *theNSNode* to be the global item node for *Page1*, and set the *addNamespace* flag to True, as shown:

```
dataNode.GetReferenceEx(vbNullString, pageGlobalNode, Nothing, True)
```

The function would return the reference to the <data:info> element, but would also modify the global item node to include the unknown *data* namespaces, as shown:

```
<global sid="global" xmlns:data="URI2">
```

You could then store the reference in that global item or any of its descendants, since the namespace is now properly defined.

Example

In the following example, a page node is passed to the . The then uses and to locate the last item node in the page. is then called to get the reference to that node, which is returned to the caller.

The following sample function receives any form node and returns a reference to the last page of the form. First, the function uses **GetNodeType** to determine if the supplied node is a the form level. If so, the function uses **GetChildren** to locate the first page node. If not, the function uses **GetParent** to locate a page level node. The function then uses **GetNext** to locate the last page in the form and calls **GetReferenceEx** to retrieve a reference to that node.

```
Function GetLastPage(Node)

   Dim MainNode, TestNode ' objects

   Set MainNode = Node

   ' Locate the page level node that is the nearest ancestor or child of
   ' the current node.  If the node is a form node, get the child.  If the
   ' node is any other type, get the parent until a page node is
   ' retrieved.

   If MainNode.getNodeType = UFL_FORM Then
      Set MainNode = MainNode.GetChildren
   Else

      ' Locate the page node of the form by iterating through the parents
      ' of the supplied node.  At the end of this loop, MainNode will be
      ' a page node.

      Do While (Not(MainNode.GetNodeType = UFL_PAGE))
         Set MainNode = MainNode.GetParent
      Loop
   End If

   ' Set TestNode to be the MainNode (the page node) before beginning the
   ' next loop.

   Set TestNode = MainNode

   ' Locate the last page node by iterating through the siblings of the
   ' global page node.  At the end of this loop, TestNode will be Nothing
   ' and MainNode will be the last page node.

   Do While (Not(TestNode Is Nothing))
      Set MainNode = TestNode
      Set TestNode = TestNode.GetNext
   Loop

   ' Get a reference to the MainNode and return that reference to the
   ' caller.

   GetLastPage = MainNode.GetReferenceEx(vbNullString, Nothing, Nothing,_
       False)

End Function
```

GetSecurityEngineName

Description

This function returns the name of the appropriate security engine for a given button or signature node. This is useful for determining which validation call you need to make to validate the signature.

Function

```
Function GetSecurityEngineName(
   theOperation As Long
   ) As String
```

Parameters

Expression	Type	Description
theOperation	**Long**	The operation you want the security engine for. Possible values are: **SEOPERATION_SIGN** — the engine is needed to sign the form. **SEOPERATION_VERIFY** — the engine is needed to verify the signature. **SEOPERATION_LISTIDENTITIES** — the engine is needed to generate a list of valid certificates for signing.

Returns

A string containing the name of the security engine on success, or throws an exception if an error occurs. The possible names are:

- CryptoAPI
- Netscape
- ClickWrap
- HMAC-ClickWrap
- PenOp

Example

The following function uses **GetSecurityEngineName** to determine which engine to use to verify the signature. It then calls the appropriate verification function. Note that the verification functions are not themselves API functions, but would use functions within the API such as **VerifySignature** and **ValidateHMACWithSecret**.

```
Function CheckSignatureByType(SigNode)

   Dim EngineName, Status  ' String

   ' Get the name of the signature engine that can verify the signature

   EngineName = SigNode.GetSecurityEngineName(SEOPERATION_VERIFY)

   ' If the signature engine is HMAC, call ValidateHMACSig, otherwise call
   ' CheckSignature.
```

```
   If EngineName = "HMAC-ClickWrap" Then
      Status = ValidateHMACSig(SigNode)
   Else
      Status = CheckSignature(SigNode)
   End If

   ' Return the signature status

   CheckSignatureByType = Status

End Function
```

GetSigLockCount

Description

This function returns the signature lock count of a node. If 0 is returned, the node is not signed by any digital signature, but it may have descendants that are signed.

Function

```
Function GetSigLockCount() As Integer
```

Parameters

There are no parameters for this function.

Returns

The number of locks on the the given node or throws an exception if an error occurs.

Example

The following function uses **GetSigLockCount** to determine whether a node has been signed. If so, the function returns True; otherwise, the function returns False.

```
Function CheckSigned(Node)

   ' Check to see if the node has a any signature locks. If so, return
   ' True to show that is signed; otherwise, return false.

   If Node.GetSigLockCount > 0 Then
      CheckSigned = True
   Else
      CheckSigned = False
   End If

End Function
```

GetSignature

Description

This function returns signature object for a given *button* or *signature* item.

Function

```
Function GetSignature() As ISignature
```

Parameters

There are no parameters for this function.

Returns

A signature object if the call is successful, or throws an exception if an error occurs.

Example

The following function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name. Finally, it uses **ValidateHMACWithSecret** to determine if the HMAC signature is valid, and returns ″Valid″ or ″Invalid″, as appropriate.

```
Function ValidateHMACSig(Form)

   Dim SigObject, XFDL  ' Objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, CommonName, SigItemRef  ' Strings
   Dim Validation  ' Integer
   Dim TempNode, SigNode  ' IFormNodeP

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get the name of the signature item

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Get available server certificates for Generic RSA signing

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
      ' vbNull

   ' Locate the certificate that has a common name of "User1-CP.02.01".
   ' This is the certificate we will use when verifying the signature.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "User1-CP.02.01" Then
         Set SigningCert = Cert
      End If

   Next

   ' Get the signature object from the signature node

   Set SigObject = SigNode.GetSignature
```

```
    ' Get the signer's name from the signature object

    SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
       False, 1) ' vbNull

    ' Include code that matches the signer's identity to a shared secret,
    ' and sets SharedSecret to match. In most cases, this would be a
    ' database lookup. For the purposes of this example, we will simply
    ' assign a value to SharedSecret.

    SharedSecret = "secret"

    ' Validate the signature

    Validation = SigNode.ValidateHMACWithSecret(SharedSecret, _
       SigningCert, 1) ' vbNull

    ' Check the validation code and return either "Valid" or "Invalid"

    If Validation = UFL_DS_OK Then
       ValidateHMACSig = "Valid"
    Else
       ValidateHMACSig = "Invalid"
    End If

End Function
```

GetSignatureVerificationStatus

Description

When called, this function checks to see if the digital signatures in a given form are valid.

Function

```
Function GetSignatureVerificationStatus() As Integer
```

Parameters

There are no parameters for this function.

Returns

An Integer having one of the following values:

Code	Status
UFL_SIGS_OK	The signatures are valid.
UFL_SIGS_NOTOK	One or more signatures are broken.
UFL_SIGS_UNVERIFIED	One or more signatures are unverifiable.

On error, the function throws an exception.

Example

The following function checks to see if all of the signatures in the form are valid by calling **GetSignatureVerificationStatus**. This relies on a flag that was set when

the form was first read, and does not return the current status of the signatures. If the signatures were valid when the form was read, the function returns ″Valid″; otherwise, the functions returns ″Invalid″.

```
Function CheckSignatures(Form)

   Dim TempNode  ' objects
   Dim SigStatus  ' Integer

   Set TempNode = Form

   ' Check to see if the signatures were valid when the form was read.

   SigStatus = TempNode.GetSignatureVerificationStatus
   ' If the signatures are not valid, then return "Invalid".  If the
   ' signatures are valid, then return "Valid".

   If (Not(SigStatus = 0)) Then  ' 0 = UFL_SIGS_OK
      CheckSignatures = "Invalid"
   Else
      CheckSignatures = "Valid"
   End If

End Function
```

GetType

Description

This function retrieves the type of a node. Possible types include *global*, *page*, and all item names (such as action, button, and so on).

Function

Function GetType() As **String**

Parameters

There are no parameters for this function.

Returns

A string containing the type of the node or throws an exception if an error occurs. If the type is empty or does not exist, the function returns **null**.

Example

The following function locates the first field in a single page form. First, the function uses **GetChildren** and **GetNext** to locate the first item in the first page. The function then uses **GetNext** to iterate through each item in the page, testing each item with **GetType** to determine whether it is a field. Finally, the function returns the field node, or a **null** node if no field is found.

```
Function LocateFirstField(Form)

   Dim TempNode  ' object

   Set TempNode = Form

   ' Find the first item on the first page of the form.

   Set TempNode = TempNode.GetChildren
```

```
    Set TempNode = TempNode.GetNext
    Set TempNode = TempNode.GetChildren

    ' Iterate through the items until the first field is located or an
    ' empty node is returned.

    Do While (Not(TempNode.GetType = "field")) And (Not(TempNode is _
       Nothing))
       Set TempNode = TempNode.GetNext
    Loop

    ' Return the node.  If no field node is found, a null node is returned.

    Set LocateFirstField = TempNode

End Function
```

IsSigned

Description

This function determines whether a node is signed.

Function

```
Function IsSigned(
   excludeSelf As Boolean
   ) As Boolean
```

Parameters

Expression	Type	Description
excludeSelf	**Boolean**	A signature node is always self-signed. To determine whether a second signature has been applied to that node, you must exclude the self-signing from this check. To exclude the self-signing from the signature check, set this to True. To include the self-signing, set this to False.

Returns

True if the node is signed, *False* if it is not.

Example

The following function locates the value node for a Date field, checks to see if it is signed, and sets the value if the node is not signed.

```
Sub SetDateValue(theDate, TheForm)

    Dim TempNode ' object

    ' Locate the value option for the Date field

    Set TempNode = TheForm.DereferenceEx(vbNullString, _
       "PAGE1.Date.value", 0, UFL_OPTION_REFERENCE, Nothing)

    ' Check the value node to see if it is signed. If it is not signed,
    ' set it to the value passed into the function.
```

```
   If TempNode.IsSigned(False) = False Then
      TempNode.SetLiteralEx vbNullString, theDate
   End If

End Sub
```

IsValidFormat

Description

This function returns the boolean result of whether a string is valid according to the setting of the node's *format* option.

This function does not support XForms nodes.

Function

```
Function IsValidFormat(
   theString As String
   ) As Boolean
```

Parameters

Expression	Type	Description
theString	**String**	A string to be checked against the format. For example, to check 23.2 against a specific format, the string would be "23.2".

Returns

True if the string does match the format, *False* if it does not.

Example

The following function locates the Currency field and checks to see if "23.2" conforms to the format required by the field's *format* option.

```
Sub CheckFormat(TheForm)

   Dim theItem ' object

   ' Locate the Currency field

   Set theItem = TheForm.DereferenceEx(vbNullString, _
      "PAGE1.Currency", 0, UFL_OPTION_REFERENCE, Nothing)

   ' Check the string to see if it is valid. If it is valid, then
   ' enter the string into the Currency field.
   ' Otherwise, do nothing.

   If theItem.IsValidFormat("23.2") = True Then

   theItem.SetLiteralByRefEx vbNullString, "value", 0, vbNullString, Nothing, _
      "23.2"

   End If

End Sub
```

IsXFDL

Description

This function determines whether a node belongs to the XFDL namespace.

Each namespace is defined in the form by a namespace declaration, as shown:

```
xmlns:xfdl="http://www.ibm.com/xmlns/prod/XFDL/7.0"
xmlns:custom="http://www.ibm.com/xmlns/prod/XFDL/Custom"
```

Each namespace declaration defines both a prefix and a URI for the namespace. In this sample, the prefix for the XFDL namespace is *xfdl* and the URI is *http://www.ibm.com/xmlns/prod/XFDL/7.0*.

Tags within the form are assigned specific namespaces by using the defined prefix. For example, to declare that an option was in the custom namespace you would use the prefix *custom* as shown:

```
<field sid="testField">
   <custom:custom_option>value</custom:custom_option>
</field>
```

Function

```
Function IsXFDL() As Boolean
```

Parameters

There are no parameters for this function.

Returns

True if the node belongs to the XFDL namespace, **False** if it does not, or throws an exception if an error occurs.

Example

The following function uses **GetChildren** and **GetNext** to walk through an entire form and deletes all information that is not in the XFDL namespace. The function uses **IsXFDL** to determine which nodes are in the XFDL namespace. Note that the function assumes that you are passing the root node of the form on the first call. Subsequent calls occur through recursion, which may provide any level of node.

```
Sub DeleteCustomInfo(Node)

   Dim MainNode, TestNode  ' objects
   Dim TempInt

   ' Set the MainNode to be the child of the provided node.

   Set MainNode = Node.GetChildren

   ' Use recursion to step through each node in the form.  This routine
   ' walks to the last node in each page, then traverses back up the tree,
   ' deleting nodes that are not in the XFDL namespace as it goes.

   Do While (Not(MainNode Is Nothing))
      Set TestNode = MainNode.getNext
      DeleteCustomInfo(MainNode)
      Set MainNode = TestNode
   Loop
```

```
    ' Check to see if the node passed to the routine is in the XFDL
    ' namespace.  If not, delete the node.

    If Node.isXFDL = False Then
       Node.Destroy
    End If

End Sub
```

RemoveAttribute

Description

This function removes a specific attribute from a node. For example, the following XFDL represents a value node:

```
<value custom:myAtt="x"></value>
```

To remove the custom attribute from this node, you would use *removeAttribute*.

Function

```
Sub RemoveAttribute(
   theNamespaceURI As String,
   theAttribute As String)
```

Parameters

Expression	Type	Description
theNamespaceURI	**String**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**String**	The local name of the attribute. For example, *compute*, *encoding*, and so on.

Returns

Nothing or throws an exception if an error occurs.

Notes

Attributes and the Null Namespace

If an attribute is on a node in a non-XFDL namespace, and that attribute has no namespace prefix, then the attribute is in the *null* namespace. For example, the following node is the custom namespace, as is the first attribute, but since the second attribute does not have a namespace prefix, it is in the null namespace:

```
<custom:processing custom:stage="2" user="tjones">
```

When an attribute is the null namespace, you may either provide a null value for the namespace URI or use the namespace URI for the containing element.

For example, to indicate *user* attribute on the *processing* node, you could use the null namespace or the custom namespace URI.

Attributes and Namespace Prefixes

If you refer to an attribute with a namespace prefix, **RemoveAttribute** first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Example

The following function uses **DereferenceEx** to locate a custom node in the form. It then uses **RemoveAttribute** to delete the ″stage″ attribute from the node, and calls **SetAttribute** to update the value of the status attribute to ″completed″.

```
Sub CompletedProcessing(Form)

   Dim TempNode  ' object

   Set TempNode = Form

   ' Dereference the custom processing node in the global item.

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
      "global.global.custom:processing", 0, UFL_OPTION_REFERENCE, _
       Nothing)

   ' Remove the "stage" attribute from the node.

   TempNode.RemoveAttribute vbNullString, "stage"

   ' Update the status attribute to "completed".

   TempNode.SetAttribute vbNullString, "status", "completed"
End Sub
```

RemoveEnclosure

Description

This function will either remove an enclosure from a specific datagroup or delete the enclosure from the form. Call this function on the **IFormNodeP** that contains the enclosure you want to remove.

Function

```
Sub RemoveEnclosure(
   theDataGroup As String)
```

Parameters

Expression	Type	Description
theDataGroup	**String**	The datagroup that contains the enclosed item. If null, the item will be removed from all datagroups. If an item no longer belongs to any datagroups, it is deleted from the form.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Example

The following function locates an attachment in a form an deletes it. First, the function uses **DereferenceEx** to locate the node that contains the attachment. Next, the function calls **RemoveEnclosure** to delete the attachment.

```
Sub DeleteAttachment(Form)

   Dim TempNode  ' object

   Set TempNode = Form

   ' Locate the PAGE3.DATA1 item, which contains the enclosure

   Set TempNode = TempNode.DereferenceEx(vbNullString, "PAGE3.DATA1", _
      0, UFL_ITEM_REFERENCE, Nothing)

   ' Delete the enclosure

   TempNode.RemoveEnclosure vbNullString

End Sub
```

ReplaceXFormsInstance

Description

This function either inserts or replaces an XForms instance in a form's data model. The instance can come from either from either a file or a memory block.

Call this function on the root node of the form or an instance node.

Use caution when calling this function. It can be used to overwrite signed instance data.

Note: This function automatically updates the XForms data model.

Function

```
Sub ReplaceXformsInstance(
   theModelID As String,
   theNodeRef As String,
   theNSNode As IFormNodeP,
   theFilename As String,
   theMemoryBlock As Variant
   replaceRef As Boolean)
```

Parameters

Expression	Type	Description
theModelID	**String**	The ID of the affected model. You must use vbnullstring to use the default model.
theNodeRef	**String**	An XPath reference to the instance (or portion of an instance) you want to replace. An empty string indicates the default instance of the selected model.
theNSNode	**IFormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use nothing if the node that this function is operating on has inherited the necessary namespaces.
theFilename	**String**	The file to read the instance from.
theMemoryBlock	**Variant**	The memory block that contains the instance if you are not reading from a file, input stream, or Reader. Use **nothing** if *theFilename* is used.
replaceRef	**Boolean**	If True, the node specified by *theNodeRef* is replaced with data. If False, the data is appended as the last child of the instance node.

Returns

Nothing if call is successful or throws an exception if an error occurs. .

Example

The following example shows a routine that replaces an XForms instance.

```
Sub SaveDataInstance(Form)

   Form.ReplaceXFormsInstance "model1",
      "instance('instance1')loanrecord.user_personal_info",
       Nothing, "c:\InstanceData.xml", Nothing, True

End Sub
```

SetActiveForComputationalSystem

Description

This function sets whether the computational system is active. When active, all computes in the form are evaluated on an on-going basis. When inactive, no computes are evaluated.

Note that turning the computational system on causes all computes in the form to be re-evaluated, which can be time consuming.

Function

```
Sub SetActiveForComputationalSystem(
   activeFlag As Boolean)
```

Parameters

Expression	Type	Description
activeFlag	**Boolean**	Set to True to activate the compute system or False to deactivate the compute system.

Returns

Nothing or throws an exception if an error occurs.

Example

The following function uses **SetActiveForComputationalSystem** to turn the compute system off before making a series of changes to a form. Once the changes are complete, **SetActiveForComputationalSystem** is called again to turn the compute systems back on, allowing the form to update itself based on the changes that were made.

```
Sub ProcessForm(Form)

   ' Turn the compute system off. This allows us to make many changes to
   ' the form quickly without the compute system slowing down the
   ' processing.

   Form.SetActiveForComputationalSystem(False)

   ' Call a function that adds a series of new items to the form.

   AddInformation(Form)

   ' Call a function that removes a series of items from the form.

   RemoveInformation(Form)

   ' Call a function that updates some information in the form.

   UpdateInformation(Form)

   ' Turn the compute system back on so that the form can update itself.

   Form.SetActiveForComputationalSystem(True)

End Sub
```

SetAttribute

Description

This function sets the value of a specific attribute for a node. For example, the following XFDL represents a value node:

```
<value custom:myAtt="x"></value>
```

To change the custom attribute, you would use **setAttribute**. If the attribute does not already exist, **setAttribute** will create it and assign the appropriate value.

Note: Do not use **SetAttribute** to set the compute attribute. Instead, use **SetFormula**.

Function

```
Sub SetAttribute(
   theNamespaceURI As String,
   theName As String,
   theValue As String)
```

Parameters

Expression	Type	Description
theNamespaceURI	**String**	The namespace URI for the attribute. For example: `http://www.ibm.com/xmlns/prod/XFDL/7.0`
theAttribute	**String**	The local name of the attribute. For example, *encoding*.
theValue	**String**	The value to assign to the attribute.

Returns

Nothing or throws an exception if an error occurs.

Notes

Attributes and the Null Namespace

If an attribute is on a node in a non-XFDL namespace, and that attribute has no namespace prefix, then the attribute is in the *null* namespace. For example, the following node is the custom namespace, as is the first attribute, but since the second attribute does not have a namespace prefix, it is in the null namespace:

```
<custom:processing custom:stage="2" user="tjones">
```

When an attribute is the null namespace, you may either provide a null value for the namespace URI or use the namespace URI for the containing element.

For example, to indicate *user* attribute on the *processing* node, you could use the null namespace or the custom namespace URI.

Attributes and Namespace Prefixes

If you refer to an attribute with a namespace prefix, **SetAttribute** first looks for a complete match, including both prefix and attribute name. If it does not find such a match, it will look for a matching attribute name that has no prefix but whose containing element has the same namespace.

For example, assume that the *custom* namespace and the *test* namespace both resolve to the same URI. In the following case, looking for the *id* attribute would locate the second attribute (test:id), since it has an explicit namespace declaration:

```
<a xmlns:custom="ABC" xmlns:test="ABC">
   <custom:myElement id="1" test:id="2">
</a>
```

However, in the next case, the *id* attribute does not have an explicit namespace declaration. Instead, it inherits the custom namespace. However, since the inherited namespace resolves to the same URI, the *id* attribute is still located:

```
<custom:myElement id="1">
```

Example

The following function uses **DereferenceEx** to locate a custom node in the form. It then uses **RemoveAttribute** to delete the ″stage″ attribute from the node, and calls **SetAttribute** to update the value of the status attribute to ″completed″.

```
Sub CompletedProcessing(Form)

   Dim TempNode  ' object

   Set TempNode = Form

   ' Dereference the custom processing node in the global item.

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
      "global.global.custom:processing", 0, UFL_OPTION_REFERENCE, _
       Nothing)

   ' Remove the "stage" attribute from the node.

   TempNode.RemoveAttribute vbNullString, "stage"

   ' Update the status attribute to "completed".

   TempNode.SetAttribute vbNullString, "status", "completed"

End Sub
```

SetFormula

Description

This function sets the formula for a node.

Function

```
Sub SetFormula(
   theComputation As String)
```

Parameters

Expression	Type	Description
theFormula	**String**	The formula to assign to the *aNode*. If null, the formula is assigned as null.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Example

The following function sets the formula in a field so that it will copy a value from another item in the form. First, the function uses **DereferenceEx** to locate the field. The function then calls **SetFormula** to set the field to copy the value of another field.

```
Sub AddFormula(Form)

   Dim TempNode  ' object

   Set TempNode = Form
```

```
      ' Locate the value node for the NameLabel on page 1.

      Set TempNode = TempNode.DereferenceEx(vbNullString, _
         "PAGE1.NameLabel.value", 0, UFL_OPTION_REFERENCE, Nothing)

      ' Set a formula for the node that will copy the value of the NameField
      ' into the NameLabel.

      TempNode.SetFormula "NameField.value"

End Sub
```

SetLiteralByRefEx

Description

This function finds a particular IFormNodeP as specified by a reference string. Once the IFormNodeP is found, its literal will be set as specified. If the IFormNodeP does not exist, this function will create it, but only if the IFormNodeP would be an option or argument node.

If necessary, this function can create several nodes at once. For example, if you set the literal for the second argument of an *itemlocation*, this function will create the *itemlocation* option node and the two argument nodes and then set the literal for the second argument node.

This function cannot create a IFormNodeP at the form, page, or item level; to do so, use **Create**.

The node you call this function on is used as the starting point for the search.

Note: It is not necessary to call this function when you are using XForms. The ReplaceXFormsInstance and ExtractXFormsInstancefunctions perform this task automatically.

Function

```
Sub SetLiteralByRefEx(
   theScheme As String,
   theReference As String,
   theReferenceCode As Long,
   theCharSet As String,
   theNSNode As IFormNodeP,
   theLiteral As String)
```

Parameters

Expression	Type	Description
theScheme	**String**	Reserved. This must be null.
theReference	**String**	A string that contains the reference.
theReferenceCode	**Long**	Reserved. Must be 0.
theCharSet	**String**	The character set in which *theLiteral* parameter is written. . Use null for ANSI/Unicode. Use **Symbol** for Symbol.

Expression	Type	Description
theNSNode	**IFormNodeP**	A node that is used to resolve the namespaces in *theReference* parameter (see "Determining Namespace" on page 93). Use null if the node that you are calling this function on has inherited the necessary namespaces.
theLiteral	**String**	The string that will be assigned to the literal. If null, any existing literal is removed.

Returns

Nothing if the call is successful or throws an exception if an error occurs.

Notes

IFormNodeP

Before you decide which IFormNodeP to use this function on, be sure you understand the following:

1. The IFormNodeP you supply can never be more than one level in the hierarchy above the level at which your reference string starts. For example, if the reference string begins with an option, then the IFormNodeP can be no higher in the hierarchy than an item.
2. If the IFormNodeP is at the same level or lower in the hierarchy than the starting point of the reference string, the function will attempt to locate a common ancestor. The function will locate the ancestor of the IFormNodeP that is one level in the hierarchy above the starting point of the reference string. The function will then attempt to follow the reference string back down through the hierarchy. If the reference string cannot be followed from the located ancestor (for example, if the ancestor is not common to both the IFormNodeP and the reference string), the function will fail. For example, given a IFormNodeP that represents ″field_1″ and a reference of ″field_2″, the function will access the ″page″ node above ″field_1″, and will then try to locate ″field_2″ below that node. If the two fields were not on the same page, the function would fail.

Creating a Reference String

For more information about creating a reference, see "References" on page 6.

Digital Signatures

Do not set a node that is digitally signed. Doing so will break the digital signature and produce an error.

Determining Namespace

In some cases, you may want to use the **SetLiteralByRefEx** function to set the value for a node that does not have a globally defined namespace. For example, consider the following form:

```
<label sid="Label1">
   <value>Field1.processing:myValue</value>
</label>
```

```
<field sid="Field1" xmlns:processing="URI">
   <value></value>
   <processing:myValue>10<processing:myValue>
</field>
```

In this form, the *processing* namespace is declared in the *Field1* node. Any elements within *Field1* will understand that namespace; however, elements outside of the scope of *Field1* will not.

In cases like this, you will often start your search at a node that does not understand the namespace of the node you are trying to locate. For example, you might want to locate the node referenced in the value of *Label1*. In this case, you would first locate the *Label1* value node and get its literal. Then, from the *Label1* value node, you would attempt to locate the *processing:myValue* node as shown:

```
Label1Node.SetLiteralByRefEx(vbNullString, "Field1.processing:myValue",
   0, vbNullString, vbNullString, "20")
```

In this example, the **setLiteralByRef** function would fail. The function cannot properly resolve the *processing* namespace because this namespace is not defined for the *Label1* value node. To correct this, you must also provide a node that understands the *processing* namespace (in this case, any node in the scope of *Field1*) as a parameter in the function:

```
Label1Node.SetLiteralByRefEx(vbNullString, "Field1.processing:myValue",
   0, vbNullString, Field1Node, "20")
```

Example

The following example adds a label to a form. A node is passed into the function, which then uses **GetLiteralByRefEx** to read the value of a field. The function then uses **DereferenceEx** to locate the field node, and creates a label node as a sibling using **Create**. Finally, the function creates a value for the new label node using **SetLiteralByRef**.

```
Sub AddLabel(Form)

   Dim TempNode, XFDL ' objects
   Dim Name ' strings

   Set TempNode = Form

   ' Get the value of the NameField.value option node

   Name = TempNode.GetLiteralByRefEx(vbNullString, _
      "PAGE1.NameField.value", 0, vbNullString, Nothing)

   ' Locate the NameField item node in the first page of the form

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
   "PAGE1.NameField", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get an XFDL object

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")

   ' Create a label.  This label is created as a sibling of the NameField,
   ' and is named NameLabel.

   Set TempNode = XFDL.Create(TempNode, UFL_AFTER_SIBLING, "label", _
      vbNullString, vbNullString, "NameLabel")

   ' Create a value option for the label.  This option is assigned the
   ' value of Name (as read from the field)
```

```
    TempNode.SetLiteralByRefEx vbNullString, "value", 0, vbNullString, _
       Nothing, Name

End Sub
```

SetLiteralEx

Description

This function sets the literal of a node. You should only set the literal for option or argument nodes.

Note: It is not necessary to call this function when you are using XForms. The ReplaceXFormsInstance and ExtractXFormsInstancefunctions perform this task automatically.

Function

```
Sub SetLiteralEx(
   theCharSet As String,
   theLiteral As String)
```

Parameters

Expression	Type	Description
theCharSet	**String**	The character set in which *theLiteral* parameter is written. . Use null for ANSI/Unicode. Use **Symbol** for Symbol.
theLiteral	**String**	The literal to assign to the node. If null, any existing literal is removed.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Notes

Digital Signatures

Do not set the literal of a node that has already been signed. Doing so will break the digital signature and produce an error.

Example

The following function copies the value from one field to another. First, the function uses **DeferenceEx** to locate the value node of a field on page one and reads the value using **GetLiteral**. Next, the function locates a value node on page two and writes the literal to that field using **SetLiteralEx**.

```
Sub CopyValue(Form)

   Dim TempNode  ' object
   Dim theName  ' string

   Set TempNode = Form

   ' Locate the NameField on page 1
```

```
    Set TempNode = TempNode.DereferenceEx(vbNullString, _
       "PAGE1.NameField.value", 0, UFL_OPTION_REFERENCE, Nothing)

    ' Get the literal from the value node

    theName = TempNode.GetLiteral

    ' Locate the NameField on page 2

    Set TempNode = TempNode.DereferenceEx(vbNullString, _
       "PAGE2.NameField.value", 0, UFL_OPTION_REFERENCE, Nothing)

    ' Write the literal that was read from the first value node

    TempNode.SetLiteralEx vbNullString, theName

End Sub
```

SignForm

Description

This function takes a button node and creates a digital signature for that button. The signature is created using the signature filter in the button and the private key of the signer.

Function

```
Function SignForm(
   signer As ICertificate,
   theInfo As IStringDictionary,
   theStatus As Long
   ) As ISignature
```

Parameters

Expression	Type	Description
theSigner	**ICertificate**	The certificate to use to create the signature.
theInfo	**IStringDictionary**	Always use a null value.
theStatus	**Long**	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A signature object if the call is successful, or throws an exception if an error occurs.

Example

The following function uses **DereferenceEx** to locate a signature button in the form. It then calls **GetCertificateList** to get a list of valid certificates for that

button. The function then loops through the available certificates, using **GetDataByPath** to check the common name of each certicate. When it finds the certificate with the common name of ″TJones″, it calls **SignForm** and uses that certificate to sign the form.

```
Sub ApplySignature(Form)

   Dim SigNode, SigObject  ' objects
   Dim TheCerts  ' CertificateList
   Dim CommonName  ' String
   Dim Cert  ' ICertificate

   ' Get the SignatureButton node

   Set SigNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.SignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get available certificates for that button

   Set TheCerts = SigNode.GetCertificateList(vbNullString, 1) 'vbNull

   ' Test each of the available certificates to see if it has a common
   ' name of "TJones".  If it does, use that certificate to sign
   ' the form.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      Response.Write CommonName & vbCrLf
      If CommonName = "TJones" Then
         Set SigObject = SigNode.SignForm(TheCerts(1), Nothing, 1)
            ' vbNull
      End If
   Next

End Sub
```

UpdateXFormsInstance

Description

This function supplants replaceXFormsInstance. It allows developers to insert data anywhere within the XForms instance data, or replace it entirely. The instance can come from either from either a file or a memory block. The UpdateXFormsInstance function automatically updates the XForms data model.

Call this function on the root node of the form or an instance node.

Use caution when calling this function. It can be used to overwrite signed instance data.

Note: This function automatically updates the XForms data model.

Function

```
Sub UpdateXformsInstance(
   theModelID As String,
   theNodeRef As String,
   theNSNode As IFormNodeP,
   theFilename As String,
   theMemoryBlock As Variant,
   updateType As Long)
```

Parameters

Expression	Type	Description
theModelID	**String**	The ID of the affected model. You must use vbnullstring to use the default model.
theNodeRef	**String**	An XPath reference to the instance (or portion of an instance) you want to insert data into or replace. An empty string indicates the default instance of the selected model.
theNSNode	**IFormNodeP**	A node that inherits the namespaces used in the reference. This node defines the namespaces for the function. Use nothing if the node that this function is operating on has inherited the necessary namespaces.
theFilename	**String**	The file to read the instance data from. Note that if both a file and a memory block are provided, the file will take precedence.
theMemoryBlock	**Variant**	The memory block that contains the instance if you are not reading from a file, input stream, or Reader. Use **nothing** if *theFilename* is used.
updateType	**Long**	Indicates which type of update to perform: **XFORMS_UPDATE_REPLACE** — Replaces the specified data element. **XFORMS_UPDATE_APPEND** — Adds the data to the end of the specified data instance or element as a child element. **XFORMS_UPDATE_INSERT_BEFORE** — Adds the data as a sibling of the specified element. This sibling is placed before the specified element.

Returns

Nothing if call is successful or throws an exception if an error occurs. .

Example

The following example shows a routine that replaces an XForms instance.

```
Sub SaveDataInstance(Form)

   Form.UpdateXFormsInstance "model1",
      "instance('instance1')loanrecord.user_personal_info",
       Nothing, "c:\InstanceData.xml", Nothing, True, XFORMS_UPDATE_REPLACE)

End Sub
```

ValidateHMACWithSecret

Description

This function determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the signer's shared secret to use this function. For Signature Pad signatures, you may use this function without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This function will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **VerifySignature** function to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Function

```
Function ValidateHMACWithSecret(
   theSecret As String,
   theServerCert As ICertificate,
   theStatus As Long
   ) As Integer
```

Parameters

Expression	Type	Description
theSecret	**String**	The shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters. For example, if the secrets were ″blue″ and ″red″, you would pass ″bluered″ to the function. If there is no shared secret pass an empty string.
theServerCert	**ICertificate**	The server certificate. If the HMAC signature is valid, the function will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using VerifySignature. If you pass null, the function will simply validate the HMAC signature.

Expression	Type	Description
theStatus	**Long**	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

:A constant if the verification is successful, or throws an exception if an error occurs. The following table lists the possible return values:

Code	Numeric Value	Status
UFL_DS_OK	0	The signature is verified.
UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
UFL_DS_NOSIGNATURE	13526	There is no signature.
UFL_DS_NOTAUTHENTICATED	1240	The signer cannot be authenticated.
UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name. Finally, it uses **ValidateHMACWithSecret** to determine if the HMAC signature is valid, and returns ″Valid″ or ″Invalid″, as appropriate.

```
Function ValidateHMACSig(Form)

   Dim SigObject, XFDL  ' Objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, CommonName, SigItemRef  ' Strings
   Dim Validation  ' Integer
```

```
   Dim TempNode, SigNode  ' IFormNodeP

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get the name of the signature item

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Get available server certificates for Generic RSA signing

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
      ' vbNull

   ' Locate the certificate that has a common name of "User1-CP.02.01".
   ' This is the certificate we will use when verifying the signature.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "User1-CP.02.01" Then
         Set SigningCert = Cert
      End If
   Next

   ' Get the signature object from the signature node

   Set SigObject = SigNode.GetSignature

   ' Get the signer's name from the signature object

   SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
      False, 1) ' vbNull

   ' Include code that matches the signer's identity to a shared secret,
   ' and sets SharedSecret to match. In most cases, this would be a
   ' database lookup. For the purposes of this example, we will simply
   ' assign a value to SharedSecret.

   SharedSecret = "secret"

   ' Validate the signature

   Validation = SigNode.ValidateHMACWithSecret(SharedSecret, _
      SigningCert, 1) ' vbNull

   ' Check the validation code and return either "Valid" or "Invalid"

   If Validation = UFL_DS_OK Then
      ValidateHMACSig = "Valid"
   Else
      ValidateHMACSig = "Invalid"
   End If

End Function
```

ValidateHMACWithHashedSecret

Description

This function determines whether an HMAC signature is valid. HMAC signatures include both Authenticated Clickwrap and Signature Pad signatures.

For Authenticated Clickwrap signatures, you must know the hash of the signer's shared secret to use this function. For Signature Pad signatures, you may use this function without the shared secret if the signature was created without one. In any case, the shared secret should be available from a corporate database or other system.

This function will also notarize (that is, digitally sign) a valid HMAC signature if you provide a digital certificate. However, notarization will not occur if the signature does not include a shared secret. Once notarized, you must use the **VerifySignature** function to validate the signature.

Note: Authenticated Clickwrap is a separately licensed product. Please ensure that your company has the license to use Authenticated Clickwrap before you provide forms or functionality that rely on it.

Function

```
Function ValidateHMACWithHashedSecret(
   hashedSecret As Variant,
   theServerCert As ICertificate,
   theStatus As Long
   ) As Integer
```

Parameters

Expression	Type	Description
hashedSecret	**Variant**	The hash of the shared secret that identifies the user. This should be available from a corporate database or other system. If there is more than one shared secret, you must concatenate the strings with no separating characters and then hash the combined secret. For example, if the secrets were ″blue″ and ″red″, you would pass the hash of ″bluered″ to the function. If there is no shared secret, pass and empty string. You must encode the byte array as follows: **Authenticated Clickwrap (HMAC)** UTF-8 **Signature Pad** UTF-16LE The method for doing this depends on the software library you are using to interface with the COM API. Note that the function expects the hashed secret to be a single-byte binary array. Using a double-byte binary array produces an incorrect result.
theCertificate	**ICertificate**	The server certificate. If the HMAC signature is valid, the function will use the private key of this certificate to digitally sign the HMAC signature. This signature is appended to the signature item, and can be verified using UFLVerifySignature. If you pass null, the function will simply validate the HMAC signature.
theStatus	**Long**	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A constant if the verification is successful, or throws an exception if an error occurs. The following table lists the possible return values:

Code	Numeric Value	Status
UFL_DS_OK	0	The signature is verified.
UFL_DS_ALGORITHM UNAVAILABLE	13590	The appropriate verification engine for the signature is not available.
UFL_DS_F2MATCHSIGNER	13529	The certificate does not match the signer's name.
UFL_DS_FAILED AUTHENTICATION	1272	The signature is invalid or the secret used is incorrect.
UFL_DS_HASHCOMPFAILED	13527	The document has been tampered with.
UFL_DS_NOSIGNATURE	13526	There is no signature.
UFL_DS_NOT AUTHENTICATED	1240	The signer cannot be authenticated.
UFL_DS_UNEXPECTED	13589	An unexpected error occurred.
UFL_DS_UNVERIFIABLE	859	The signature cannot be verified.

Example

The following function validates an HMAC signature using a hashed secret. First, the function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name and **Hash** to create a hashed secret. Finally, it uses **ValidateHMACWithHashedSecret** to determine if the HMAC signature is valid, and returns ″Valid″ or ″Invalid″, as appropriate.

Note that this example also relies on a second function called **StringToBinary**. This function converts a string to a single-byte binary array, which is required for the hash function. This prevents COM from converting the string to a double-byte array before hashing it, which would produce an incorrect result.

```
Function ValidateHMACSigHashed(Form)

   Dim SigObject, XFDL, HashObject, SecurityManager  ' objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, HashedSecret, CommonName, _
      SigItemRef  ' Strings
   Dim Validation  ' Integer
   Dim TempNode, SigNode  ' IFormNodeP

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get the name of the signature item

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)
```

```
   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Get available server certificates for Generic RSA signing

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
      ' vbNull

   ' Locate the certificate that has a common name of "User1-CP.02.01".
   ' This is the certificate we will use when verifying the signature.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "User1-CP.02.01" Then
         Set SigningCert = Cert
      End If
   Next

   ' Get the signature object from the signature node

   Set SigObject = SigNode.GetSignature

   ' Get the signer's name from the signature object

   SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
      False, 1) ' vbNull

   ' Include code that matches the signer's identity to a shared secret
   ' that is hashed, and sets SharedSecret to match. In most cases, this
   ' would be a database lookup. For the purposes of this example, we will
   ' use the Hash function to assign a hashed value to HashedSecret.
   ' Get the Security Manager object

   Set SecurityManager = _
      CreateObject("PureEdge.security_SecurityManager")

   ' Get the Hash object

   Set HashObject = SecurityManager.LookupHashAlgorithm("sha1")

   ' Set the Hashed secret. First convert the secret to a single-byte
   ' binary array, then hash the secret.

   SharedSecret = StringToBinary("secret")
   HashedSecret = HashObject.Hash(SharedSecret)

   ' Validate the signature

   Validation = SigNode.ValidateHMACWithHashedSecret(HashedSecret, _
      SigningCert, 1) ' vbNull

   ' Check the validation code and return either "Valid" or "Invalid"

   If Validation = UFL_DS_OK Then
      ValidateHMACSigHashed = "Valid"
   Else
      ValidateHMACSigHashed = "Invalid"
   End If
End Function

' The following function is required to convert a string to a single-byte binary
'   array before hashing that string. This prevents COM from converting
'  the string to a multi-byte format, which would produce and incorrect
```

```
'  hash.

Function StringToBinary(String)

   Dim Counter, Binary

   For Counter = 1 to len(String)
      Binary = Binary & ChrB(Asc(Mid(String, Counter, 1)))
   Next

   StringToBinary = Binary

End Function
```

VerifyAllSignatures

Description

This function verifies the correctness of all digital signatures in a given form whose root node is provided. It finds all items of type signature and calls **VerifySignature** for each signature. Errors are logged for all invalid signatures.

This function checks the following conditions for each signature:

- The signature item contains mimedata.
- The mimedata contains a hash value and signer certificate.
- The signer certificate contains the same ID as that recorded in the signature item's *signer* option.
- The signer certificate has not expired.

Function

```
Function VerifyAllSignatures(
   reportAsErrorsFlag As Boolean
   ) As Integer
```

Parameters

Expression	Type	Description
reportAsErrorsFlag	**Boolean**	Set to True if you want errors about the signatures to be reported by throwing an exception, or False if you want the error code to be only returned through the return value.

Returns

An integer having one of the following values:

Code	Status
UFL_SIGS_OK	The signatures are valid.
UFL_SIGS_NOTOK	One or more signatures are broken.
UFL_SIGS_UNVERIFIED	One or more signatures are unverifiable.

If one or more of the signatures is not valid and the *reportAsErrorsFlag* is *true*, an exception is thrown. On error, the function throws an exception.

Example

The following example uses **VerifyAllSignatures** to check all of the signatures in the form, then returns ″Valid″ if the signatures are okay or ″Invalid″ if they are not.

```
Function CheckAllSignatures(Form)

   Dim Status  ' Integer

   Status = Form.VerifyAllSignatures(False)
   If Status = UFL_SIGS_OK Then
      CheckAllSignatures = "Valid"
   Else
      CheckAllSignatures = "Invalid"
   End If

End Function
```

VerifySignature

Description

This function verifies the correctness of the given digital signature. You supply the root of the form that contains the signature you want to verify. This function checks the following conditions:

- The signature item contains mimedata.
- The mimedata contains a hash value and signer certificate.
- The signer certificate contains the same ID as that recorded in the signature item's *signer* option.
- The signer certificate has not expired.

A plain text representation of the form (filtered by the signature item's filter) is constructed and the result is hashed. This hash value must match the hash value stored in the signature.

Function

```
Function VerifySignature(
   signatureItem As IFormNodeP,
   theCertChain As String,
   reportAsErrorsFlag As Boolean
  ) As Integer
```

Parameters

Expression	Type	Description
signatureItem	**IFormNodeP**	The signature to verify.
theCertChain	**String**	Reserved. Must be null.
reportAsErrorsFlag	**Boolean**	Set to True if you want errors about the signatures to be reported by throwing an exception or False if you want the error code to be returned through the return value.

Returns

A **Long** having one of the following values, depending on the status of the signature:

Code	Status
UFL_DS_OK	The signature is verified.
UFL_DS_ALGORITHMUNAVAILABLE	The appropriate verification engine for the signature is not available.
UFL_DS_CERTEXPIRED	The certificate has expired.
UFL_DS_CERTNOTFOUND	The certificate cannot be located.
UFL_DS_CERTNOTTRUSTED	The certificate is not trusted.
UFL_DS_CERTREVOKED	The certificate has been revoked.
UFL_DS_CRLINVALID	The certificate revocation list is invalid.
UFL_DS_F2MATCHSIGNER	The certificate does not match the signer's name.
UFL_DS_HASHCOMPFAILED	The document has been tampered with.
UFL_DS_ISSUERCERTEXPIRED	The issuer's certificate has expired.
UFL_DS_ISSUERINVALID	The issuer is invalid for the certificate used to sign.
UFL_DS_ISSUERKEYUSAGE UNACCEPTABLE	The issuer certificate's key usage extension does not match what the key was used for.
UFL_DS_ISSUERNOTCA	The certificate's issuer is not a Certificate Authority.
UFL_DS_ISSUERNOTFOUND	The issuer's certificate was not located.
UFL_DS_ISSUERSIGFAILED	Verification of the issuer's certificate failed.
UFL_DS_KEYREVOKED	The key used to create the signature has been revoked.
UFL_DS_KEYUSAGEUNACCEPTABLE	The certificate's key usage extension does not match what the key was used for.
UFL_DS_KRLINVALID	The Key Revocation List is invalid.
UFL_DS_NOSIGNATURE	There is no signature.
UFL_DS_NOTAUTHENTICATED	The signer cannot be authenticated.
UFL_DS_POLICYUNACCEPTABLE	The certificate's policy extension does not match the acceptable policies.
UFL_DS_SIGNATUREALTERED	The signature has been tampered with.
UFL_DS_UNEXPECTED	An unexpected error occurred.
UFL_DS_UNVERIFIABLE	The signature cannot be verified.

If the signature is not valid and the *reportAsErrorsFlag* is *True*, an exception is thrown. On error, the function throws an exception.

Example

The following function checks to see the signature in the form is valid. First, the function uses **DereferenceEx** to locate the signature button. It then uses **GetLiteralByRefEx** to get the name of the signature item, and uses another **DereferenceEx** to locate that item. Next, it uses **VerifySignature** to determine whether the signature is valid. If so, it return the string ″Valid″. If not, it uses **DeleteSignature** to delete the signature and returns the string ″Invalid″.

```
Function CheckSignature(Form)

   Dim TempNode, SigNode  ' objects
   Dim SigStatus  ' Integer
   Dim SigItemRef  ' Strings

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
      "PAGE1.SignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get a reference to the signature item from the signature option

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Verify the signature

   SigStatus = Form.VerifySignature(SigNode, vbNullString, False)

   ' If the signature is not verified, then delete the signature and set
   ' the return code to "Invalid".  Otherwise, set the return code to
   ' "Valid".

   If (Not(SigStatus = UFL_DS_OK)) Then
      TempNode.DeleteSignature SigNode
      CheckSignature = "Invalid"
   Else
      CheckSignature = "Valid"
   End If
End Function
```

WriteForm

Description

This function will write a form to the specified file . Call this function on the root node of the form. The version number of the form determines the format of the output file. You can specify whether to compress the output file and whether to observe the *transmit* and *save* settings in the form.

If no format is specified, the default is to write the form in the same format in which it was read. If the form in question was created dynamically by your application, **WriteForm** will, by default, write it as an XFDL form in uncompressed format.

Function

```
Sub WriteForm(
   theFilePath As String,
   triggerItem As IFormNodeP,
   flags As Long)
```

Parameters

Expression	Type	Description
theFilePath	**String**	This is the path to the file on the local disk to which the form will be written.
triggerItem	**IFormNodeP**	This is the item that caused the form to be submitted. Set to null if the API receives the form in a manner other than transmission.
flags	**Long**	The following flags are valid: **UFL_TRANSMIT_ALLOW** allows the transmit options (that is, *transmitdatagroups*, *transmitgroups*, *transmititemrefs*, *transmititems*, *transmitoptionrefs*, *transmitpagerefs* and *transmitoptions*) to control which portions of the form are sent. Without this flag, the entire form will be sent regardless of the *transmit* options in the form. **UFL_SAVE_ALLOW** allows the *saveformat* option to specify what format the form should be saved in. If no format is specified then the form will be saved in the same format that it is read. **Note:** Specify 0 if you do not want to enable any of the transmit options.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following example uses **WriteForm** to write the form in memory to a file on the local drive.

```
Sub SaveForm(Form)

   ' Write the form to a file on disk

   Form.WriteForm "c:\testform.xfd", Nothing, 0
End Sub
```

WriteFormToASPResponse

Description

This function will write a form to the ASP response object. Call this function on the root node of the form. The version number of the form determines the format of the output file. You can specify whether to compress the output file and whether to observe the *transmit* and *save* settings in the form.

If no format is specified, the default is to write the form in the same format in which it was read. If the form in question was created dynamically by your application, **WriteFormToASPResponse** will, by default, write it as an XFDL form in uncompressed format.

Function

```
Sub WriteFormToASPResponse(
   triggerItem As IFormNodeP,
   flags As Long)
```

Parameters

Expression	Type	Description
triggerItem	**IFormNodeP**	This is the item that caused the form to be submitted. Set to **null** if the API receives the form in a manner other than transmission.
flags	**Long**	The following flags are valid: **UFL_TRANSMIT_ALLOW** allows the transmit options (that is, *transmitdatagroups*, *transmitgroups*, *transmititemrefs*, *transmititems*, *transmitoptionrefs*, *transmitpagerefs* and *transmitoptions*) to control which portions of the form are sent. Without this flag, the entire form will be sent regardless of the *transmit* options in the form. **UFL_SAVE_ALLOW** allows the *saveformat* option to specify what format the form should be saved in. If no format is specified then the form will be saved in the same format that it is read. **Note:** Specify 0 if you do not want to enable any of the transmit options.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following example uses **WriteFormToASPResponse** to write the form in memory to the ASP response object.

```
Sub RespondWithForm(Form)

   Form.WriteFormToASPResponse Nothing, 0

End Sub
```

XMLModelUpdate

Description

This function updates the XML data model in the form. This is necessary if computes have changed the structure of the data model in some way, such as changing or adding bindings. These sorts of changes do not take effect until the **XMLModelUpdate** function is called.

Function

Sub XMLModelUpdate()

Parameters

There are no parameters for this function.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following example uses **SetLiteralByRefEx** to change a binding in the form, so that it binds to a different option. It then calls **XMLModelUpdate** so that the data model reflects the change.

```
Function ChangeBinding(Form)

   ' Change the binding to a field on the second page.

   Form.SetLiteralByRefEx vbNullString, _
      "global.global.xmlmodel.bindings[0][boundoption]", 0, _
      vbNullString, Nothing, "PAGE2.NameField.value"

   ' Update the XML model.

   Form.XMLModelUpdate

End Function
```

The Hash Functions

The **Hash** functions allow you to hash messages.

- To use the Hash functions you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

Hash

Description

This function hashes a message using the hashing algorithm of your choice.

Function

```
Function Hash(
   theMessage As Variant)
```

Parameters

Expression	Type	Description
theMessage	**Variant**	The message you want to hash. Note that the function expects the hashed secret to be a single-byte binary array. Using a double-byte binary array will produce an incorrect result.

Returns

A hashed message, or throws an exception if an error occurs.

Example

The following function validates an HMAC signature using a hashed secret. First, the function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name and **LookupHashAlgorith** and **Hash** to create a hashed secret. Finally, it uses **ValidateHMACWithHashedSecret** to determine if the HMAC signature is valid, and returns ″Valid″ or ″Invalid″, as appropriate.

Note that this example also relies on a second function called **StringToBinary**. This function converts a string to a single-byte binary array, which is required for the hash function. This prevents COM from converting the string to a double-byte array before hashing it, which would produce an incorrect result.

```
Function ValidateHMACSigHashed(Form)

   Dim SigObject, XFDL, HashObject, SecurityManager  ' objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, HashedSecret, CommonName, _
      SigItemRef  ' Strings
```

```
Dim Validation  ' Integer
Dim TempNode, SigNode  ' IFormNodeP

Set TempNode = Form

' Get the SignatureButton node

Set TempNode = Form.DereferenceEx(vbNullString, _
   "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

' Get the name of the signature item

SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
   0, vbNullString, Nothing)

' Get the signature item node

Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
   UFL_ITEM_REFERENCE, Nothing)

' Get available server certificates for Generic RSA signing

Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
   ' vbNull

' Locate the certificate that has a common name of "User1-CP.02.01".
' This is the certificate we will use when verifying the signature.

For Each Cert in TheCerts
   CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
      False, 1) ' vbNull
   If CommonName = "User1-CP.02.01" Then
      Set SigningCert = Cert
   End If
Next

' Get the signature object from the signature node

Set SigObject = SigNode.GetSignature

' Get the signer's name from the signature object

SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
   False, 1) ' vbNull

' Include code that matches the signer's identity to a shared secret
' that is hashed, and sets SharedSecret to match. In most cases, this
' would be a database lookup. For the purposes of this example, we will
' use the Hash function to assign a hashed value to HashedSecret.
' Get the Security Manager object

Set SecurityManager = _
   CreateObject("PureEdge.security_SecurityManager")

' Get the Hash object

Set HashObject = SecurityManager.LookupHashAlgorithm("sha1")

' Set the Hashed secret. First convert the secret to a single-byte
' binary array, then hash the secret.

SharedSecret = StringToBinary("secret")
HashedSecret = HashObject.Hash(SharedSecret)

' Validate the signature
```

```
      Validation = SigNode.ValidateHMACWithHashedSecret(HashedSecret, _
         SigningCert, 1) ' vbNull

      ' Check the validation code and return either "Valid" or "Invalid"
      If Validation = UFL_DS_OK Then

         ValidateHMACSigHashed = "Valid"
      Else
         ValidateHMACSigHashed = "Invalid"
      End If

   End Function

   ' The following function is required to convert a string to a single-byte '
   ' binary array before hashing that string. This prevents COM from converting
   ' the string to a multi-byte format, which would produce an incorrect hash.

Function StringToBinary(String)

      Dim Counter, Binary

      For Counter = 1 to len(String)
         Binary = Binary & ChrB(Asc(Mid(String, Counter, 1)))
      Next

      StringToBinary = Binary

   End Function
```

The Initialization Functions

The initialization function provides an easy function for initializing the API.

- To use the Initialization function you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

- To use the Initialization function, you must first create the PureEdge.DTK object, as shown:

```
Set DTK = CreateObject("PureEdge.DTK")
```

  You can then call the function on this object.

IFSInitialize

Description

This function initializes the API. The parameters specify which version of the API your application should bind with (see the Notes below for more details).

You must call this function before calling any of the other functions in the API.

Function

```
Sub IFSInitialize(
   progName As String,
   progVer As String,
   apiVer As String)
```

Parameters

Expression	Type	Description
progName	**String**	The name of the application calling **IFSInitialize**. This name is used to identify the application within the .ini file. It also sets the name that is returned by the XFDL **applicationName** function.
progVer	**String**	The version number of the application calling **IFSInitialize**. If the .ini file has an entry for this version of the application, the application will bind to the version of the API listed in that entry.
apiVer	**String**	The version number of the API the application should use by default. If the .ini file does not contain an entry for the specific application, the application will bind to the API specified by this parameter.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Notes

About Binding Your Applications to the API

When you initialize the API, the IFSInitialize function determines which version of the API to use based on the parameters you pass it. This allows you to exercise a great deal of control over which version of the API is used by your applications, and prevents the problems normally associated with common DLL files (often referred to as ″DLL hell″).

IFSInitialize uses a configuration file to determine which version of the API will bind to any application. This allows multiple versions of the API to co-exist on your computer, and ensures that your applications use the correct version of the API.

The configuration file is called PureEdgeAPI.ini and is installed with the API. Refer to the *IBM Workplace Forms Server — API Installation and Setup Guide* for the exact location of the file.

Note: You should redistribute the PureEdgeAPI.ini file with any applications that use the API. See the IBM Workplace Forms Server — API Installation and Setup Guide for more information about redistributing applications.

The configuration file contains a section for each application that might call the API, plus a default ″API″ section. Each section contains a list of version numbers in the following format:

```
<version of application> = <folder containing appropriate version of API>
```

For example, the configuration file might look like this:

```
[API]
5.1.0 = 51
5.0.0 = 50
[CustomApplication]
1.1.0 = 51
1.0.0 = 50
```

In this case, the folder indicated on the right hand side of each statement is part of the relative path to the API, and assumes the API was installed in the default folder. For example, under Windows ″50″ would resolve to:

```
c:\WinNT\System32\PureEdge\50
```

You can also specify an absolute path by placing a drive letter before the path. For example, ″c:\50″ would resolve to:

```
c:\50\
```

When you initialize the API, you include three parameters in the initialization call:

- The name of your application (as it would appear in the configuration file).
- The version of your application.
- The version of the API that your application should bind to by default.

The initialization call will first check the configuration file to see if your application is listed. For example, using the configuration file above, if you make an initialization call for ″CustomApplication″ version ″1.1.0″, then the application binds to the API in the ″51″ folder.

If your application is not listed in the configuration file, the initialization call uses the default version of the API. For example, using the configuration file above, if you declare ″5.1.0″ as the default API, then your application binds to the API in the ″51″ folder.

You can add your own entries to the configuration file before distributing it to your customers, or you can rely on the default API entries.

Note: IFSInitialize was introduced for version 4.5.0 of the API. Binding does not work in this manner for earlier versions of the API. Do not include earlier versions of the API in the configuration file.

Example

In the example below, **IFSInitialize** initializes the API for the application called **aspApp.**

```
Function LoadForm(FileName)

   Dim DTK, XFDL, TempForm ' objects

   ' Get a DTK object and initialize the API

   Set DTK = CreateObject("PureEdge.DTK")
   DTK.IFSInitialize "aspApp", "1.0.0", "2.6.0"

   ' Get an XFDL object and read the form from the supplied file

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TempForm = XFDL.ReadForm(FileName, 0)

   ' Return the form

   Set LoadForm = TempForm

End Function
```

IFSInitializeWithLocale

Description

This function initializes the API. The parameters specify the default locale, and which version of the API your application should bind with (see the Notes below for more details).

You must call this function before calling any of the other functions in the API.

Function

```
Sub IFSInitializeWithLocale(
   progName As String,
   progVer As String,
   apiVer As String
   theLocale As String)
```

Parameters

Expression	Type	Description
progName	**String**	The name of the application calling **IFSInitializeWithLocale**. This name is used to identify the application within the .ini file. It also sets the name that is returned by the XFDL **applicationName** function.

Expression	Type	Description
progVer	**String**	The version number of the application calling **IFSInitializeWithLocale**. If the .ini file has an entry for this version of the application, the application will bind to the version of the API listed in that entry.
apiVer	**String**	The version number of the API the application should use by default. If the .ini file does not contain an entry for the specific application, the application will bind to the API specified by this parameter.
theLocale	**String**	The default locale of the application.

Returns

Nothing if call is successful or throws an exception if an error occurs.

Notes

About Binding Your Applications to the API

When you initialize the API, the **IFSInitializeWithLocale** function determines which version of the API to use based on the parameters you pass it. This allows you to exercise a great deal of control over which version of the API is used by your applications, and prevents the problems normally associated with common DLL files (often referred to as ″DLL hell″).

IFSInitializeWithLocale uses a configuration file to determine which version of the API will bind to any application. This allows multiple versions of the API to co-exist on your computer, and ensures that your applications use the correct version of the API.

The configuration file is called PureEdgeAPI.ini and is installed with the API. Refer to the *IBM Workplace Forms Server — API Installation and Setup Guide* for the exact location of the file.

Note: You should redistribute the PureEdgeAPI.ini file with any applications that use the API. See the IBM Workplace Forms Server — API Installation and Setup Guide for more information about redistributing applications.

The configuration file contains a section for each application that might call the API, plus a default ″API″ section. Each section contains a list of version numbers in the following format:

```
<version of application> = <folder containing appropriate version of API>
```

For example, the configuration file might look like this:

```
[API]
2.6.1 = 70
2.6.0 = 70
[CustomApplication]
1.1.0 = 70
1.0.0 = 70
```

In this case, the folder indicated on the right hand side of each statement is part of the relative path to the API, and assumes the API was installed in the default folder. For example, under Windows “70” would resolve to:

```
C:\Program Files\IBM\Workplace Forms\Server\26\API\redist
   \msc32\PureEdge\26
```

You can also specify an absolute path by placing a drive letter before the path. For example, “c:\70” would resolve to:

```
c:\70\
```

When you initialize the API, you include three parameters in the initialization call:

- The name of your application (as it would appear in the configuration file).
- The version of your application.
- The version of the API that your application should bind to by default.

The initialization call will first check the configuration file to see if your application is listed. For example, using the configuration file above, if you make an initialization call for “CustomApplication” version “1.1.0”, then the application binds to the API in the “70” folder.

If your application is not listed in the configuration file, the initialization call will use the default version of the API. For example, using the configuration file above, if you declare “2.6.1” as the default API, then your application binds to the API in the “70” folder.

You can add your own entries to the configuration file before distributing it to your customers, or you can rely on the default API entries.

Example

In the example below, **IFSInitializeWithLocale** initializes the API for the application called **aspApp.**

```
Function LoadForm(FileName)

   Dim DTK, XFDL, TempForm ' objects

   ' Get a DTK object and initialize the API

   Set DTK = CreateObject("PureEdge.DTK")
   DTK.IFSInitializeWithLocale "aspApp", "1.0.0",
      "2.6.0", "fr-FR"

   ' Get an XFDL object and read the form from the supplied file

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TempForm = XFDL.ReadForm(FileName, 0)

   ' Return the form

   Set LoadForm = TempForm

End Function
```

The LocalizationManager Functions

The **LocalizationManager** functions control which language the API uses to report errors.

- To use the LocalizationManager functions you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

- To use the LocalizationManager function, you must first create the PureEdge.i18n_LocalizationManager object, as shown:

```
Set theManager = CreateObject("PureEdge.i18n_LocalizationManager")
```

  You can then call the function on this object.

GetCurrentThreadLocale

Description

This function returns which *locale* is in use for the current thread. This determines what language the API uses when reporting errors. By default, the API uses the default locale.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US

Language	Locale	Locale Name
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES

Language	Locale	Locale Name
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
Sub GetCurrentThreadLocale()
```

Parameters

There are no parameters for this function.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following function calls **GetCurrentThreadLocale** to get the locale.

```
Function GetLanguage()

   Dim theManager  ' object

   Set theManager = CreateObject("PureEdge.i18n_LocalizationManager")
   GetLanguage=theManager.GetCurrentThreadLocale

End Function
```

GetDefaultLocale

Description

This function returns the default *locale* the API uses when reporting errors.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG

Language	Locale	Locale Name
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU

Language	Locale	Locale Name
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
Sub GetDefaultLocale()
```

Parameters

There are no parameters for this function.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following function calls **GetDefaultLocale** to get the locale.

```
Function GetLanguage()

   Dim theManager  ' object

   Set theManager = CreateObject("PureEdge.i18n_LocalizationManager")
   GetLanguage=theManager.GetDefaultLocale

End Function
```

SetCurrentThreadLocale

Description

This function sets which *locale* the API uses when reporting errors. By default, the API uses the application's default locale.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE

Language	Locale	Locale Name
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY

Language	Locale	Locale Name
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
Sub SetCurrentThreadLocale(
   theLocale As String)
```

Parameters

Expression	Type	Description
theLocale	**String**	The name of the locale.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following function checks the *language* string to determine which locale to use. It then calls **SetCurrentThreadLocale** to set the appropriate locale.

```
Sub SetCurrentLanguage(Language)

   Dim theManager ' object

   Set theManager = CreateObject("PureEdge.i18n_LocalizationManager")
   If Language = "English" Then
      theManager.SetCurrentThreadLocale "en_US"
   Else
      theManager.SetCurrentThreadLocale "fr_CA"
   End If

End Sub
```

SetDefaultLocale

Description

This function sets the default *locale* the API uses when reporting errors, if no other locale is specified. By default, the API uses the locale specified by the operating system.

The API supports the following locales:

Language	Locale	Locale Name
Chinese	Simplified Han, China	zh-Hans-CN
	Simplified Han, Singapore	zh-Hans-SG
	Traditional Han, Hong Kong S.A.R., China	zh-Hant-HK
	Traditional Han, Taiwan	zh-Hant-TW
Croatian	Croatia	hr-HT
Czech	Czech Republic	cs-CZ
Danish	Denmark	da-DK
Dutch	Belgium	nl-BE
	The Netherlands	nl-NL
English	Australia	en-AU
	Belgium	en-BE
	Canada	en-Ca
	Hong Kong S.A.R., China	en-HK
	India	en-IN
	Ireland	en-IE
	New Zealand	en-NZ
	Philippines	en-PH
	Singapore	en-SG
	South Africa	en-ZA
	United Kingdom	en-GB
	United States	en-US
Finnish	Finland	fi-FI
French	Belgium	fr-BE
	Canada	fr-CA
	France	fr-FR
	Luxembourg	fr-LU
	Switzerland	fr-CH
German	Austria	de-AT
	Germany	de-DE
	Luxembourg	de-LU
	Switzerland	de-CH
Greek	Greece	el-GR
Hungarian	Hungary	hu-HU
Italian	Italy	it-IT
	Switzerland	it-CH
Japanese	Japan	ja-JP
Korean	South Korea	ko-KR
Norwegian Bokmål	Norway	nb-NO
Polish	Poland	pl-PL
Portuguese	Brazil	pt-BR

Language	Locale	Locale Name
	Portugal	pt-PT
Romanian	Romania	ro-RO
Russian	Russian	ru-RU
Slovak	Slovakia	sk-SK
Slovene	Slovenia	sl_SI
Spanish	Argentina	es-AR
	Bolivia	es-BO
	Chile	es-CL
	Colombia	es-CO
	Costa Rica	es-CR
	Dominican Republic	es-DO
	Ecuador	es-EC
	El Salvador	es-SV
	Guatemala	es-GT
	Honduras	es-HN
	Mexico	es-MX
	Nicaragua	es-NI
	Panama	es-PA
	Paraguay	es-PY
	Peru	es-PE
	Puerto Rico	es-PR
	Spain	es-ES
	United States	es-US
	Uruguay	es-UY
	Venezuela	es-VE
Swedish	Sweden	sv-SE
Turkish	Turkey	tr-TR

The locale name consists of two parts: the language code and the country code, as shown

```
<language code>_<COUNTRY CODE>
```

For example, to specify the Japanese locale, you would type:

```
jp_JP
```

If you need a more specific locale, you may add additional codes after the country code. For example, to indicate French in France with a Euro dialect, you would type:

```
fr_FR_EU
```

Function

```
Sub SetDefaultLocale(
   theLocale As String)
```

Parameters

Expression	Type	Description
theLocale	**String**	The name of the locale.

Returns

Returns nothing if the call is successful, or throws an exception if an error occurs.

Example

The following function checks the *language* string to determine which locale to use. It then calls **SetDefaultLocale** to set the appropriate locale.

```
Sub SetDefaultLanguage(Language)

   Dim theManager  ' object

   Set theManager = CreateObject("PureEdge.i18n_LocalizationManager")

   If Language = "English" Then
      theManager.SetDefaultLocale "en_US"
   Else
      theManager.SetDefaultLocale "fr_CA"
   End If

End Sub
```

The SecurityManager Functions

The SecurityManager functions allow you to retrieve the Security Manager and obtain a hashing algorithm.

- To use the SecurityManager function you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

- To use the SecurityManager function, you must first create the PureEdge.security_SecurityManager object, as shown:

```
Set SecurityManager = _
   CreateObject("PureEdge.security_SecurityManager")
```

  You can then call the function on this object.

LookupHashAlgorithm

Description

This function retrieves a hash object. Use the hash object to hash shared secrets for the **ValidateHMACWithHashedSecret** function.

Function

```
Function LookupHashAlgorithm(
   algorithmName As String
   ) As IHash
```

Parameters

Expression	Type	Description
algorithmName	**String**	The name of the hash algorithm you want to retrieve. The available hash algorithms are *sha1* and *md5*.

Returns

A hash object, or throws an exception if an error occurs.

Example

The following function validates an HMAC signature using a hashed secret. First, the function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name and **LookupHashAlgorith** and **Hash** to create a hashed secret. Finally, it uses **ValidateHMACWithHashedSecret** to determine if the HMAC signature is valid, and returns ″Valid″ or ″Invalid″, as appropriate.

Note that this example also relies on a second function called **StringToBinary**. This function converts a string to a single-byte binary array, which is required for the hash function. This prevents COM from converting the string to a double-byte array before hashing it, which would produce an incorrect result.

```
Function ValidateHMACSigHashed(Form)

   Dim SigObject, XFDL, HashObject, SecurityManager  ' objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, HashedSecret, CommonName, _
      SigItemRef  ' Strings
   Dim Validation  ' Integer
   Dim TempNode, SigNode  ' IFormNodeP

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get the name of the signature item
   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Get available server certificates for Generic RSA signing

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
      ' vbNull

   ' Locate the certificate that has a common name of "User1-CP.02.01".
   ' This is the certificate we will use when verifying the signature.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "User1-CP.02.01" Then
         Set SigningCert = Cert
      End If
   Next

   ' Get the signature object from the signature node

   Set SigObject = SigNode.GetSignature

   ' Get the signer's name from the signature object

   SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
      False, 1) ' vbNull

   ' Include code that matches the signer's identity to a shared secret
   ' that is hashed, and sets SharedSecret to match. In most cases, this
   ' would be a database lookup. For the purposes of this example, we will
   ' use the Hash function to assign a hashed value to HashedSecret.
   ' Get the Security Manager object

   Set SecurityManager = _
      CreateObject("PureEdge.security_SecurityManager")

   ' Get the Hash object

   Set HashObject = SecurityManager.LookupHashAlgorithm("sha1")
   ' Set the Hashed secret. First convert the secret to a single-byte
   ' binary array, then hash the secret.
```

```
   SharedSecret = StringToBinary("secret")
   HashedSecret = HashObject.Hash(SharedSecret)

   ' Validate the signature

   Validation = SigNode.ValidateHMACWithHashedSecret(HashedSecret, _
      SigningCert, 1) ' vbNull

   ' Check the validation code and return either "Valid" or "Invalid"

   If Validation = UFL_DS_OK Then
      ValidateHMACSigHashed = "Valid"
   Else
      ValidateHMACSigHashed = "Invalid"
   End If

End Function

' The following function is required to convert a string to a single-byte '
' binary array before hashing that string. This prevents COM from converting
' the string to a multi-byte format, which would produce an incorrect hash.

Function StringToBinary(String)

   Dim Counter, Binary

   For Counter = 1 to len(String)
      Binary = Binary & ChrB(Asc(Mid(String, Counter, 1)))
   Next
   StringToBinary = Binary

End Function
```

The Signature Functions

The **Signature** functions allow you to work with signature objects.

- To use the Initialization functions you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

GetDataByPath

Description

This function retrieves a piece of data from a signature object.

Function

```
Function GetDataByPath(
   thePath As String,
   tagData As Boolean,
   encoded As Boolean
   ) As String
```

Parameters

Expression	Type	Description
thePath	**String**	The path to the data you want to retrieve. See the Notes section below for more information on data paths.
tagData	**Boolean**	True if the path should be prepended to the data, or False if not. If the path is prepended, an equals sign (=) is used as a separator. For example, suppose the path is ″Signing Cert: Issuer: CN″ and the data is ″IBM″. If True, the path will be prepended, producing ″CN=IBM″. If False, the path will not be prepended, and the result will be ″IBM″.
encoded	**Boolean**	True if the return data is base 64 encoded, or False if not. The function returns binary data in base 64 encoding.

Notes

About Data Paths

Data paths describe the location of information within a signature, just like file paths describe the location of files on a disk. You describe the path with a series of colon separated tags. Each tag represents either a piece of data, or an object that contains further pieces of data (just like directories can contain files and subdirectories).

For example, to retrieve the version of a signature, you would use the following data path:

```
Demographics
```

However, to retrieve the signer's common name, you first need to locate the signing certificate, then the subject, then finally the common name within the subject, as follows:

```
SigningCert: Subject: CN
```

Some tags may contain more than one piece of information. For example, the issuer's organizational unit may contain a number of entries. You can either retrieve all of the entries as a comma separated list, or you can specify a specific entry by using a zero-indexed element number.

For example, the following path would retrieve a comma separated list:

```
SigningCert: Issuer: OU
```

Adding an element number of 0 would retrieve the first organizational unit in the list, as shown:

```
SingingCert: Issuer: OU: 0
```

Signature Tags

The following table lists the tags available in a signature object. Note that Clickwrap and HMAC Clickwrap signatures have additional tags (detailed in Clickwrap Signature Tags and HMAC Clickwrap Tags).

Tag	Description
Engine	The security engine used to create the signature. This is an object that contains further information, as detailed in *Security Engine Tags*.
SigningCert	The certificate used to create the signature. This is an object that contains further information, as detailed in *Certificate Tags*. Note that this object does not exist for Clickwrap or HMAC Clickwrap signatures.
HashAlg	The hash algorithm used to create the signature.
CreateDate	The date on which the signature was created.
Demographics	A string describing the signature.
LastVerificationStatus	A short representing the verification status of the signature. This is updated whenever the signature is verified. See "VerifySignature" on page 107 for a complete list of the possible values.

Clickwrap Signature Tags

The following table lists additional tags available in both Clickwrap and HMAC Clickwrap signatures. Note that HMAC Clickwrap signatures have further tags (detailed in HMAC Clickwrap Tags).

Tag	Description
TitleText	The text for the Windows title bar of the signature dialog box.
MainPrompt	The text for the title portion of the signature dialog box.
MainText	The text for the text portion of the signature dialog box.

Tag	Description
Question1Text	The first question in the signature dialog box.
Answer1Text	The signer's answer.
Question2Text	The second question in the signature dialog box.
Answer2Text	The signer's answer.
Question3Text	The third question in the signature dialog box.
Answer3Text	The signer's answer.
Question4Text	The fourth question in the signature dialog box.
Answer4Text	The signer's answer.
Question5Text	The fifth question in the signature dialog box.
Answer5Text	The signer's answer.
EchoPrompt	Text that the signer must echo to create a signature.
EchoText	The signer's response to the echo text.
ButtonPrompt	The text that provides instructions for the Clickwrap signature buttons.
AcceptText	The text for the accept signature button.
RejectText	The text for the reject signature button.

Certificate Tags

The following table lists the tags available in a certificate object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Tag	Description
Subject	The subject's distinguished name. This is an object that contains further information, as detailed in Distinguished Name Tags.
Issuer	The issuer's distinguished name. This is an object that contains further information, as detailed in Distinguished Name Tags.
IssuerCert	The issuer's certificate. This is an object that contains the complete list of certificate tags.
Engine	The security engine that generated the certificate. This is an object that contains further information, as detailed in Security Engine Tags.
Version	The certificate version.
BeginDate	The date on which the certificate became valid.
EndDate	The date on which the certificate expires.
Serial	The certificate's serial number.
SignatureAlg	The signature algorithm used to sign the certificate.
PublicKey	The certificate's public key.
FriendlyName	The certificate's friendly name.

Distinguished Name Tags

The following table lists the tags available in a distinguished name object. Note that Clickwrap and HMAC Clickwrap signatures do not contain these tags.

Tag	Description
CN	The common name.
E	The e-mail address.
T	The title.
O	The organization.
OU	The organizational unit.
C	The country.
L	The locality.
ST	The state.
All	The entire distinguished name.

HMAC Clickwrap Tags

The following table lists the tags available in HMAC Clickwrap signature. Note that these tags are in addition to both the regular Signature Tags and the Clickwrap Signature Tags.

Tag	Description
HMACSigner	A string indicating which answers store the signer's ID.
HMACSecret	A string indicating which answers store the signer's secret.
Notarization	The notarizing signatures. This is one or more signature objects that contain further information, as detailed in Signature Tags . There can be any number of notarizing signatures. Use an element number to retrieve a specific signature. For example, to get the first notarizing signature use: `Notarization: 0` If no element number is provided, the data will be retrieved from the first valid notarizing signature found. If no valid notarizing signatures are found, the function will return null.

Security Engine Tags

The following table lists the tags available in the security engine object:

Tag	Description
Name	The name of the security engine used by the server.
Help	The help text for the security engine.
HashAlg	A hash algorithm supported by the security engine.

Returns

A string containing the certificate data (null if no data is found), or throws an exception if an error occurs.

Example

The following function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name. Finally, it uses **ValidateHMACWithSecret** to determine if the HMAC signature is valid, and returns "Valid" or "Invalid", as appropriate.

```
Function ValidateHMACSig(Form)

   Dim SigObject, XFDL  ' Objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, CommonName, SigItemRef  ' Strings
   Dim Validation  ' Integer
   Dim TempNode, SigNode  ' IFormNodeP

   Set TempNode = Form
   ' Get the SignatureButton node

   Set TempNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get the name of the signature item

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Get available server certificates for Generic RSA signing

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
      ' vbNull

   ' Locate the certificate that has a common name of "User1-CP.02.01".
   ' This is the certificate we will use when verifying the signature.

   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "User1-CP.02.01" Then
         Set SigningCert = Cert
      End If

   Next

   ' Get the signature object from the signature node

   Set SigObject = SigNode.GetSignature

   ' Get the signer's name from the signature object

   SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
      False, 1) ' vbNull

   ' Include code that matches the signer's identity to a shared secret,
   ' and sets SharedSecret to match. In most cases, this would be a
   ' database lookup. For the purposes of this example, we will simply
   ' assign a value to SharedSecret.
```

```
   SharedSecret = "secret"

   ' Validate the signature

   Validation = SigNode.ValidateHMACWithSecret(SharedSecret, _
      SigningCert, 1) ' vbNull

   ' Check the validation code and return either "Valid" or "Invalid"

   If Validation = UFL_DS_OK Then
      ValidateHMACSig = "Valid"
   Else
      ValidateHMACSig = "Invalid"
   End If

End Function
```

GetSigningCert

Description

This function retrieves the signing certificate from a signature object.

Function

Function GetSigningCert() As Certificate

Parameters

There are no parameters for this function.

Returns

The signing certificate.

Example

The following example gets the signing certificate from a signature object, then iterates through the certificate issuers until it reaches the end of the chain. During the iteration, each certificate is passed to a function that processes them.

```
Sub processCertChain(TheSig)

   Dim TheCert, IssuerCert  ' Variant

   ' Get the signing certificate from the signature

   Set TheCert = TheSig.GetSigningCert

   ' Loop through the certificate chain, passing each certificate to the
   ' ProcessCert function. The loop ends when the issuer certificate is
   ' Nothing.

   Do While (Not(TheCert Is Nothing))

      ' Pass the certificate to the ProcessCert function.  Note that
      ' this is not an API function, but rather a function you would
      ' write to process the certificate in some way.

      ProcessCert(TheCert)

      ' Get the issuer certificate from the TheCert

      Set IssuerCert = TheCert.GetIssuer(1) ' vbNull
```

```
        ' Assign theCert to equal the issuerCert for next iteration of the
        ' loop.

        Set TheCert = IssuerCert

    Loop

End Sub
```

The XFDL Functions

The **XFDL** functions create the root nodes of forms and handle administrative tasks related to the Form Library.

- To use the Initialization functions you must import the IFS_COM_API type library, as shown:

```
<!--  METADATA TYPE = "typelib"
      FILE = "c:\winnt\system32\IFS_COM_API.tlb"  -->
```

- To use the XFDL functions, you must first create the PureEdge.xfdl_XFDL object, as shown:

```
Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
```

  You can then call the function on this object.

Create

Description

This function creates a new IFormNodeP and attaches it to the form hierarchy at the indicated location. Once created, the type and identifier of a IFormNodeP cannot be changed.

Note that you can also use **SetLiteralByRefEx** to create an **IFormNodeP** at the option level and below. Using **SetLiteralByRefEx** is often easier and faster than using **Create**.

Function

```
Function Create(
   aNode As IFormNodeP,
   where As Long,
   theType As String,
   theLiteral As String,
   theFormula As String,
   theIdentifier As String
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
aNode	**IFormNodeP**	The new IFormNodeP is placed in the form hierarchy in relation to this node. If null, this creates a new IFormNodeP hierarchy (a new form)

Expression	Type	Description
where	Long	A constant that describes the location, in relation to the parameter *aNode*, in which the new node should be placed: **UFL_APPEND_CHILD** — adds the new node as the last child of aNode. **UFL_AFTER_SIBLING** — adds the new node as a sibling of *aNode*, placing it immediately after that node. **UFL_BEFORE_SIBLING —** adds the new node as a sibling of *aNode*, placing it immediately before that node. Note: If the parameter *aNode* is null, then this parameter should be set to 0.

theType	**String**	The type to assign to the IFormNodeP being created. This is only necessary for page and item nodes. Use null for all other nodes. The type cannot be changed after the node has been created.
		
		If you are creating a non-XFDL node, you must also include the namespace that the node should belong to, as shown:
		`<namespace prefix>:<type>`
		
		For example:
		`custom:myItem`
		
		If you do not provide a namespace, the function will assign the default namespace for the form.

theLiteral	**String**	The literal to assign to this IFormNodeP. null is valid.
theFormula	**String**	The formula to assign to this IFormNodeP. null is valid.

theIdentifier	**String**	The identifier to assign to this IFormNodeP. The identifier cannot be changed after the node has been created. null is valid.
		
		If you are creating an option or argument level node, this must also include the namespace the node should belong to. Use the following format:
		`<namespace prefix>:<type>`
		
		For example:
		`custom:myOption`
		
		If you do not provide a namespace, the function will assign the default namespace for the form.

Returns

The new **IFormNodeP** or throws an exception if an error occurs.

Example

The following example adds a label to a form. A node is passed into the function, which then uses **GetLiteralByRefEx** to read the value of a field. The function then uses **DereferenceEx** to locate the field node, and creates a label node as a sibling using **Create**. Finally, the function creates a value for the new label node using **SetLiteralByRef**.

```
Sub AddLabel(Form)

   Dim TempNode, XFDL ' objects
   Dim Name ' strings

   Set TempNode = Form

   ' Get the value of the NameField.value option node

   Name = TempNode.GetLiteralByRefEx(vbNullString, _
      "PAGE1.NameField.value", 0, vbNullString, Nothing)

   ' Locate the NameField item node in the first page of the form

   Set TempNode = TempNode.DereferenceEx(vbNullString, _
   "PAGE1.NameField", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get an XFDL object

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")

   ' Create a label.  This label is created as a sibling of the NameField,
   ' and is named NameLabel.

   Set TempNode = XFDL.Create(TempNode, UFL_AFTER_SIBLING, "label", _
      vbNullString, vbNullString, "NameLabel")

   ' Create a value option for the label.  This option is assigned the
   ' value of Name (as read from the field)

   TempNode.SetLiteralByRefEx vbNullString, "value", 0, vbNullString, _
      Nothing, Name
End Sub
```

GetEngineCertificateList

Description

This function locates all available certificates for a particular signing engine.

Function

```
Function GetEngineCertificateList(
   engineName As String,
   theStatus As Long
   ) As CertificateList
```

Parameters

Expression	Type	Description
engineName	**String**	The name of the signing engine. Valid signing engines include: Generic RSA, CryptoAPI, Netscape, and Entrust. (Note that Generic RSA is the union of CryptoAPI and Netscape.)

Expression	Type	Description
theStatus	**Long**	This is a status flag that reports whether the operation was successful. Possible values are: **SUSTATUS_OK** — the operation was successful. **SUSTATUS_CANCELLED** — the operation was cancelled by the user. **SUSTATUS_INPUT_REQUIRED** — the operation required user input, but could not receive it (for example, it was run on a server with no user).

Returns

A collection containing the list of certificates objects.

Example

The following function uses **DereferenceEx** and **GetLiteralByRefEx** to locate the signature item in a form. It then uses **GetEngineCertificateList** and **GetDataByPath** to locate a server signing certificate. Next, it uses **GetSignature** and **GetDataByPath** to get the signer's common name. Finally, it uses **ValidateHMACWithSecret** to determine if the HMAC signature is valid, and returns "Valid" or "Invalid", as appropriate.

```
Function ValidateHMACSig(Form)

   Dim SigObject, XFDL  ' Objects
   Dim TheCerts  ' CertificateList
   Dim Cert, SigningCert  ' ICertificate
   Dim SignerName, SharedSecret, CommonName, SigItemRef  ' Strings
   Dim Validation  ' Integer
   Dim TempNode, SigNode  ' IFormNodeP

   Set TempNode = Form

   ' Get the SignatureButton node

   Set TempNode = Form.DereferenceEx(vbNullString, _
      "PAGE1.HMACSignatureButton", 0, UFL_ITEM_REFERENCE, Nothing)

   ' Get the name of the signature item

   SigItemRef = TempNode.GetLiteralByRefEx(vbNullString, "signature", _
      0, vbNullString, Nothing)

   ' Get the signature item node

   Set SigNode = TempNode.DereferenceEx(vbNullString, SigItemRef, 0, _
      UFL_ITEM_REFERENCE, Nothing)

   ' Get available server certificates for Generic RSA signing

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TheCerts = XFDL.GetEngineCertificateList("Generic RSA", 1)
      ' vbNull

   ' Locate the certificate that has a common name of "User1-CP.02.01".
   ' This is the certificate we will use when verifying the signature.
```

```
   For Each Cert in TheCerts
      CommonName = Cert.GetDataByPath("SigningCert: Subject: CN", _
         False, 1) ' vbNull
      If CommonName = "User1-CP.02.01" Then
         Set SigningCert = Cert
      End If

   Next

   ' Get the signature object from the signature node

   Set SigObject = SigNode.GetSignature

   ' Get the signer's name from the signature object

   SignerName = SigObject.GetDataByPath("SigningCert: Subject: CN", _
      False, 1) ' vbNull

   ' Include code that matches the signer's identity to a shared secret,
   ' and sets SharedSecret to match. In most cases, this would be a
   ' database lookup. For the purposes of this example, we will simply
   ' assign a value to SharedSecret.

   SharedSecret = "secret"

   ' Validate the signature

   Validation = SigNode.ValidateHMACWithSecret(SharedSecret, _
      SigningCert, 1) ' vbNull

   ' Check the validation code and return either "Valid" or "Invalid"

   If Validation = UFL_DS_OK Then
      ValidateHMACSig = "Valid"
   Else
      ValidateHMACSig = "Invalid"
   End If

End Function
```

IsDigitalSignaturesAvailable

Description

This function determines whether digital signatures are available on the current computer.

Function

```
Function IsDigitalSignaturesAvailable() As Boolean
```

Parameters

There are no parameters for this function.

Returns

True if digital signatures are available on this computer; otherwise, **False**. On error, the function throws an exception.

Example

The following function calls **IsDigitalSignaturesAvailable** to deterimine whether the digital signature engine is available. If so, it returns ″Available″; otherwise, it returns ″Not Available″.

```
Function SigsAvailable()

   Dim XFDLObject  ' Object
   Dim Available  ' Boolean

   ' Get the XFDL object.

   Set XFDLObject = CreateObject("PureEdge.xfdl_XFDL")

   ' Check to see if the engine is available.

   Available = XFDLObject.IsDigitalSignaturesAvailable
   ' Return the appropriate response.

   If Available = True Then
      SigsAvailable = "Available"
   Else
      SigsAvailable = "Not Available"
   End If

End Function
```

ReadForm

Description

This function will read a form into memory from a specified file .

Function

```
Function ReadForm(
   theFilePath As String,
   flags As Long
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
theFilePath	**String**	The path to the source file on the local disk.

Expression	Type	Description

flags	**Long**	The following flags cause special behaviors. If using multiple flags, combine them using a bitwise OR. For example:
		`UFL_AUTOCOMPUTE_OFF |`
		`UFL_AUTOCREATE_FORMATS_OFF`
		
		0 — no special behavior.
		
		UFL_AUTOCOMPUTE_OFF — Reads the form into memory, but disables the compute system so that no computes are evaluated.
		
		UFL_AUTOCREATE_CONTROLLED_OFF — Reads the form into memory, but disables the creation of all options that are maintained only in memory (for example, itemnext, itemprevious, pagenext, pageprevious, and so on).
		
		UFL_AUTOCREATE_FORMATS_OFF — Reads the form into memory, but disables the evaluation of all format options.
		
		UFL_SERVER_SPEED_FLAGS — Turns off the following features: computes, automatic formatting, duplicate sid detection, the event model, and relative page and item tags (for example, itemprevious, itemnext, and so on). This setting significantly improves server processing times.
		
		UFL_XFORMS_INITIALIZE_ONLY — Turns off the following features: controlled item construction, UI connection to the XForms model, action handling set up, and the rebuild/recalculate/revalidate/refresh sequence after instance replacements.

Returns

Returns an **IFormNodeP** that is the root node of the form, or throws an exception if an error occurs.

Notes

Duplicate Scope IDs

If a form contains duplicate scope IDs (for example, two items on the same page with the same SID), **ReadForm** will fail to read the form and will return an error. This enforces correct XFDL syntax, and eliminates certain security risks that exist when duplicate scope IDs appear in signed forms.

Digital Signatures

When a form containing one or more digital signatures is read, the signatures will be verified. The result of the verification is stored in a flag that can be checked by calling **GetSignatureVerificationStatus**.

Note that this flag is only set by **ReadForm**, and its value will not be adjusted by changes made to the form after it has been read. This means that calls such as **SetLiteralEx** may actually break a signature (by changing the value of a signed item), but that this will not adjust the flags value. To verify a signature after changes have been made to a form, it is best to use **VerifyAllSignatures**.

Note that when a form is signed, all signed computes are frozen at their start value (regardless of whether the compute system is disabled).

Server-Side Processing

Using the UFL_SERVER_SPEED_FLAGS setting significantly improves performance during server-side processing. We strongly recommend you use this flag if you do not require computes to update while processing the form.

Example

In the following example, the function uses **ReadForm** to load a form into memory and return the root node of the form.

```
Function LoadForm(FileName)

   Dim DTK, XFDL, TempForm ' objects

   ' Get a DTK object and initialize the API

   Set DTK = CreateObject("PureEdge.DTK")
   DTK.IFSInitialize "aspApp", "1.0.0", "6.5.0"

   ' Get an XFDL object and read the form from the supplied file

   Set XFDL = CreateObject("PureEdge.xfdl_XFDL")
   Set TempForm = XFDL.ReadForm(FileName, 0)

   ' Return the form

   Set LoadForm = TempForm

End Function
```

ReadFormFromASPRequest

Description

This function will read a form into memory from the ASP request object.

Function

```
Function ReadFormFromASPRequest(
   flags As Long
   ) As IFormNodeP
```

Parameters

Expression	Type	Description
flags	**Long**	The following flags cause special behaviors. If using multiple flags, combine them using a bitwise OR. For example: `UFL_AUTOCOMPUTE_OFF | UFL_AUTOCREATE_FORMATS_OFF` **0** — no special behavior. **UFL_AUTOCOMPUTE_OFF** — Reads the form into memory, but disables the compute system so that no computes are evaluated. **UFL_AUTOCREATE_CONTROLLED_OFF** — Reads the form into memory, but disables the creation of all options that are maintained only in memory (for example, itemnext, itemprevious, pagenext, pageprevious, and so on). **UFL_AUTOCREATE_FORMATS_OFF** — Reads the form into memory, but disables the evaluation of all format options. **UFL_SERVER_SPEED_FLAGS** — Turns off the following features: computes, automatic formatting, duplicate sid detection, the event model, and relative page and item tags (for example, itemprevious, itemnext, and so on). This is intended to decrease server processing times. **UFL_XFORMS_INITIALIZE_ONLY** — Turns off the following features: controlled item construction, UI connection to the XForms model, action handling set up, and the rebuild/recalculate/revalidate/refresh sequence after instance replacements.

Returns

Returns an **IFormNodeP** that is the root node of the form, or throws an exception if an error occurs.

Notes

Duplicate Scope IDs

If a form contains duplicate scope IDs (for example, two items on the same page with the same SID), **ReadForm** will fail to read the form and will return an error. This enforces correct XFDL syntax, and eliminates certain security risks that exist when duplicate scope IDs appear in signed forms.

Digital Signatures

When a form containing one or more digital signatures is read, the signatures will be verified. The result of the verification is stored in a flag that can be checked by calling **GetSignatureVerificationStatus**.

Note that this flag is only set by **ReadForm**, and its value will not be adjusted by changes made to the form after it has been read. This means that calls such as **SetLiteralEx** may actually break a signature (by changing the value of a signed item), but that this will not adjust the flag''s value. To verify a signature after changes have been made to a form, it is best to use **VerifyAllSignatures**.

Note that when a form is signed, all signed computes are frozen at their start value (regardless of whether the compute system is disabled).

Example

The following example uses **ReadFormFromASPRequest** to load a form into memory and return the root node of the form.

```
   Function LoadFormFromRequestObject()
      Dim DTK, XFDL, TempForm    ' objects

      ' Get a DTK object and initialize the API

      Set DTK = CreateObject("PureEdge.DTK")
      DTK.IFSInitialize "aspApp", "1.0.0", "6.5.0"

      ' Get an XFDL object and read the form from the request object

      Set XFDL = CreateObject("PureEdge.xfdl_XFDL")

      ' Read the form into memory

      Set TempForm = XFDL.ReadFormFromASPRequest(0)

      ' Return the form

      Set LoadFormFromRequestObject = TempForm

End Function
```

Appendix. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Office 4360
One Rogers Street
Cambridge, MA 02142
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
IBM
Workplace
Workplace Forms

Intel, Intel Inside (logos), MMX, and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

C

D

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

X

Program Number: 5724-N08

Printed in USA

S229-1528-00